EXHIBIT I

US00RE39486E

(19) **United States**

(12) **Reissued Patent**

Cleron et al.

(10) **Patent Number: US RE39,486 E**

(45) **Date of Reissued Patent: Feb. 6, 2007**

(54) **EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM**

(75) Inventors: **Michael A. Cleron**, Menlo Park, CA (US); **Stephen Fisher**, Menlo Park, CA (US); **Timo Bruck**, Mountain View, CA (US)

(73) Assignee: **Apple Computer, Inc.**, Cupertino, CA (US)

(21) Appl. No.: **10/408,789**

(22) Filed: **Apr. 3, 2003**

(Under 37 CFR 1.47)

Related U.S. Patent Documents

Reissue of:

(64) Patent No.: **6,212,575**
Issued: **Apr. 3, 2001**
Appl. No.: **08/435,377**
Filed: **May 5, 1995**

(51) **Int. Cl.**
G06F 9/00 (2006.01)
G06F 9/46 (2006.01)

(52) **U.S. Cl.** **719/328**; 719/329; 709/201; 709/202; 709/203

(58) **Field of Classification Search** 719/328–329; 709/200–203
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,297,249 A * 3/1994 Bernstein et al.
5,339,430 A * 8/1994 Lundin et al.
5,481,666 A * 1/1996 Nguyen et al.
5,530,852 A * 6/1996 Meske, Jr. et al.
5,537,526 A * 7/1996 Anderson
5,548,722 A * 8/1996 Jalalian
5,581,686 A * 12/1996 Koppolu et al.
5,584,035 A * 12/1996 Duggan et al.
5,634,129 A * 5/1997 Dickinson
5,669,005 A * 9/1997 Curbow

FOREIGN PATENT DOCUMENTS

EP 0 631 456 A2 * 12/1994
GB 2 242 293 * 1/1990

OTHER PUBLICATIONS

Reinhardt, Andy, "The Network with Smarts" BYTE, Oct. 1994, pp. 51–64.*

Lippman, Stanley B., "C++ Primer" 2nd edition, Addison–Wesley, 1991, pp. 394–397.*

Potel et al; The Architecture of the Taligent System; Dr. Dobbs Journal on CD–ROM, SP 94.*

Rush, Jeff; OpenDoc; Dr. Dobb's Journal on CD–ROM, SP 94.*

Piersol, Kurt; A Close–Up of OpenDoc; AIXpert, Jun. 1994.*

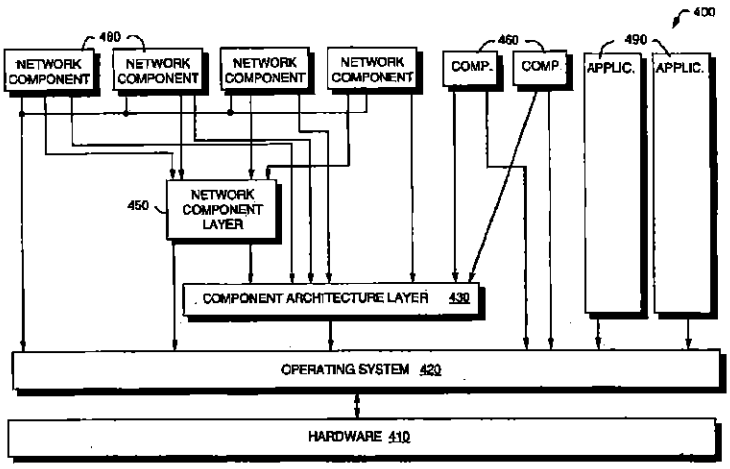
(Continued)

Primary Examiner—William Thomson
(74) *Attorney, Agent, or Firm*—Fenwick & West LLP

(57) **ABSTRACT**

An extensible and replaceable network-oriented component system provides a platform for developing networking navigation components that operate on a variety of hardware and software computer systems. These navigation components include key integrating components along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a high-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the network component system.

20 Claims, 8 Drawing Sheets



OTHER PUBLICATIONS

Schmidt et al; "An object–oriented framework for developing network server daemons", C+++ World Conference, pp. 1–15, Oct. 1993.*

"Leveraging object–oriented frameworks", Taligent white paper, 1993.*

Andert, Glerk; "Object–Frameworks in the Taligent OS", IEEE electronic Library, pp. 112–121, 1994.*

Helm et al, "Integrating information retrieval and domain specific approaches for browsing and retrieval in object–oriented class libraries", ACM Digital Library, 1991.*

Monnard et al; An object–oriented scripting environment for the WEBSs electronic book system' ACM Digital Library, 1992.*

Norr, Henry. "Cyberdog could be a breakthrough if it's Kept on a leash", MacWeek, Nov. 14, 1994, v8, n45, p. 50.*

Hess, Robert, "Cyberdog to fetch Internet Resources for Open Doc apps." MacWeek, Nov. 7, 1994, v8, n44, p. 44.*

Harkey et al, "Object component suites", Datamation, Feb. 15, 1995, v41, n3, p. 44.*

Prosise, Jeff, "Much ado about object", PC Magazine, Feb. 7, 1995, v14, n3, p. 257.*

Bonner, Paul, "Component software: putting the pieces together", Computer Shopper, Sep. 1994, v14, n9, p. 532.*

Gruman, Galen, "OpenDoc & OLE 2.0", MacWorld, Nov. '94, v11, n11, p. 96.*

Spiegel, Leo "OLE promises barrier–free computing", InfoWorld, Mar. 6, '95, v17, n10, p. 53.*

Develop, The Apple Technical Journal, "Building an OpenDoc Part Handler", Issue 19, Sep. 1994, pp. 6–16.*

S.H. Goldberg and J.A. Mounton, Jr. A Base for Portable Communications Software, IBM Systems Journal, vol. 30 (1991) No. 3, Armonk, NY, pp. 259–279.*

E.C. Arnold and D.W. Brown, Object Oriented Software Technologies Applied to Switching System Architecture and Software Development Processes, AT&T Bell Laboratories, Naperville, IL, vol. II, pp. 97–106.*

* cited by examiner

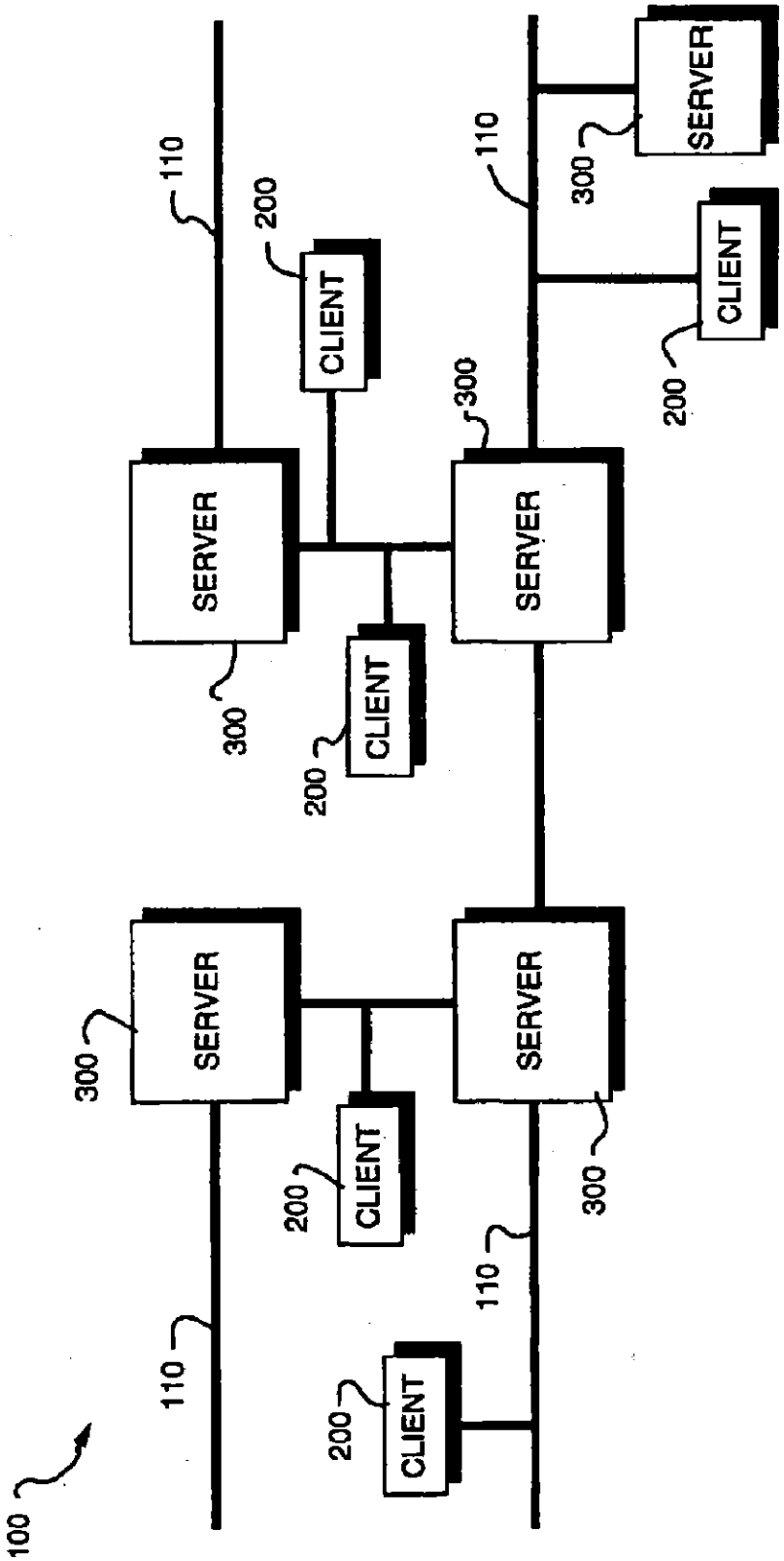


FIG. 1

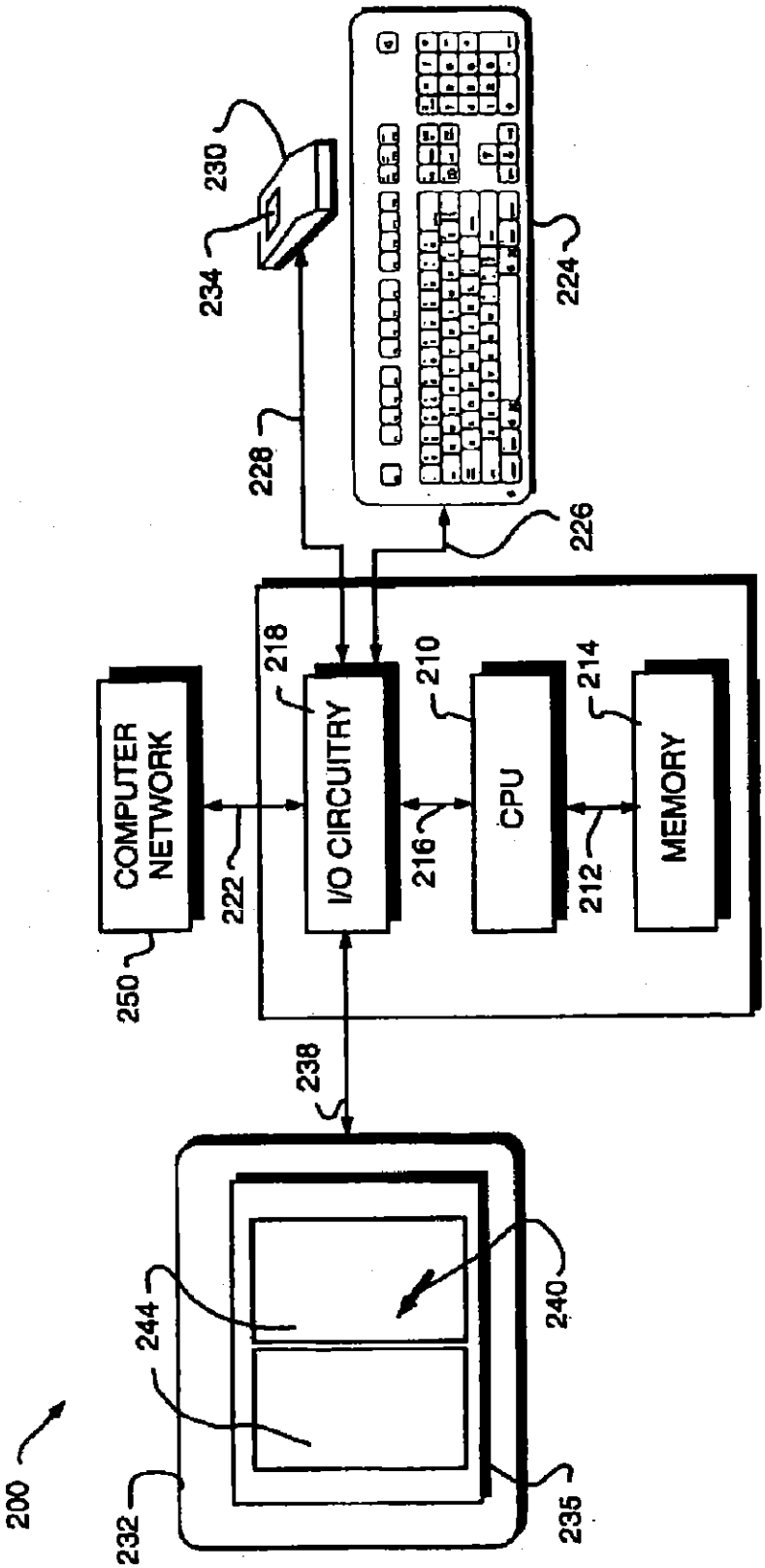

230
234
224
228
226
218
210
214
COMPUTER NETWORK
I/O CIRCUITRY
CPU
MEMORY
222
216
212
250
238
240
244
200
232
235

FIG. 2

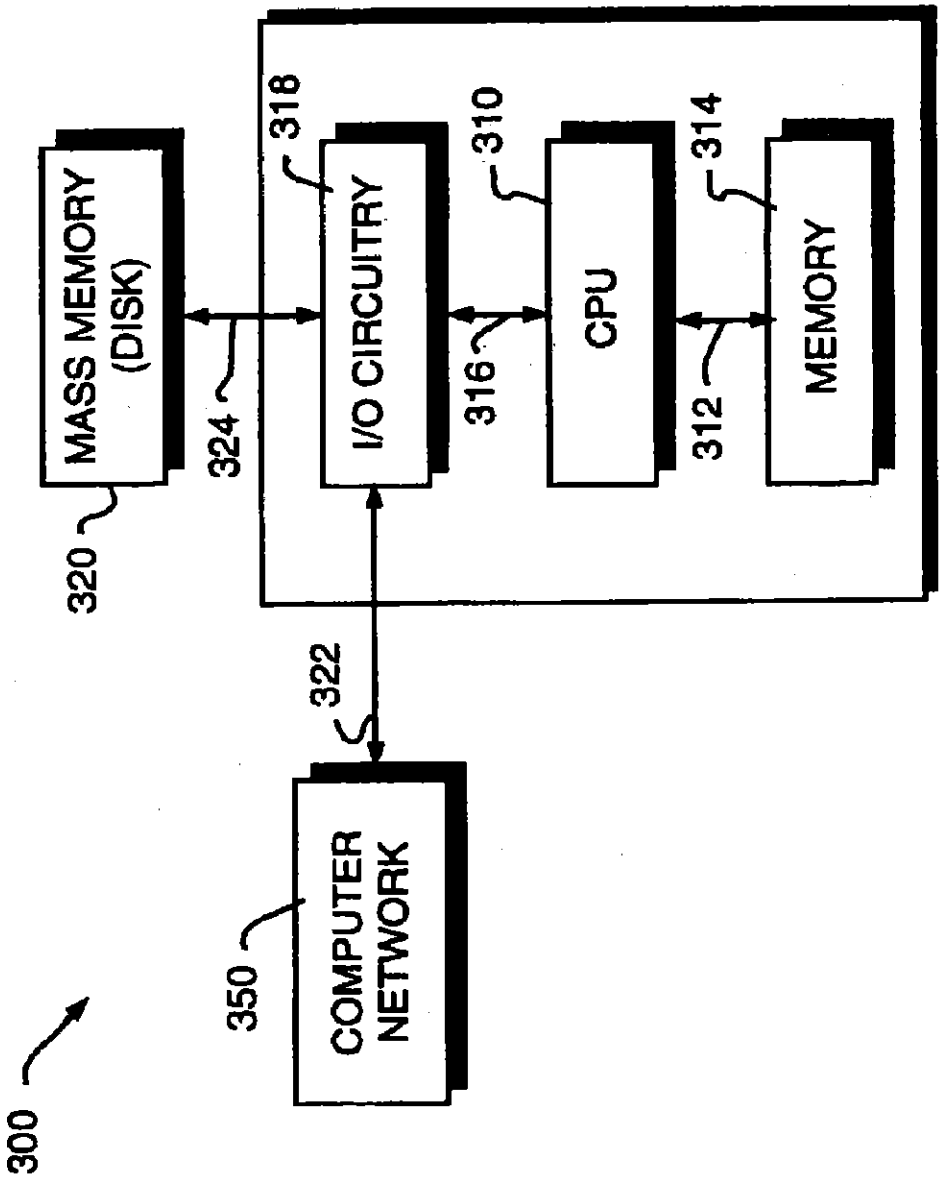

MASS MEMORY (DISK)
I/O CIRCUITRY
CPU
MEMORY
COMPUTER NETWORK
318
310
314
316
312
324
320
322
350
300

FIG. 3

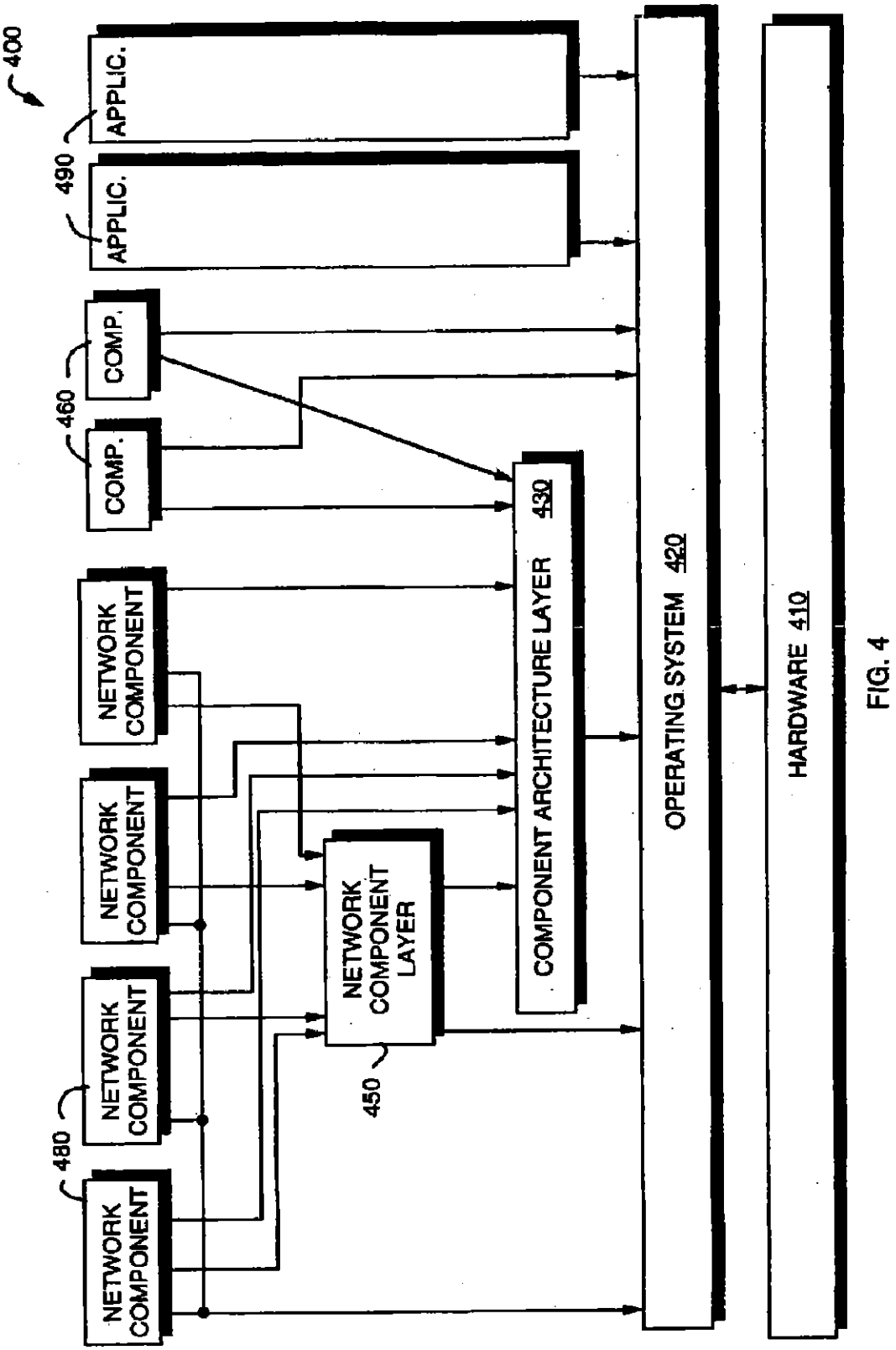

400
APPLIC.
490
APPLIC.
460
COMP.
COMP.
NETWORK COMPONENT
NETWORK COMPONENT
NETWORK COMPONENT
480
NETWORK COMPONENT
NETWORK COMPONENT LAYER
450
COMPONENT ARCHITECTURE LAYER 430
OPERATING SYSTEM 420
HARDWARE 410

FIG. 4

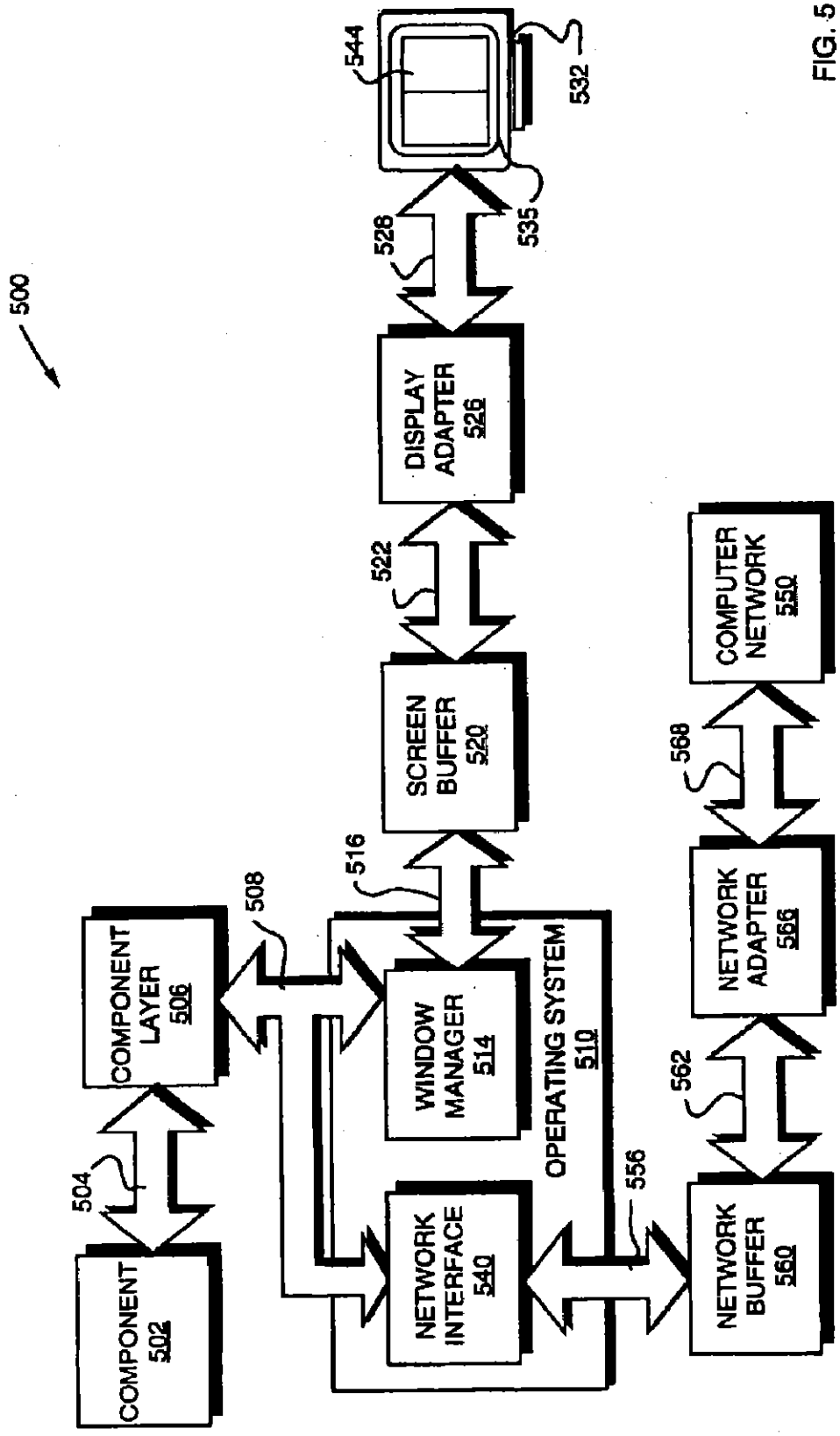

500
544
532
535
526
DISPLAY ADAPTER 526
522
SCREEN BUFFER 520
516
508
COMPONENT LAYER 506
WINDOW MANAGER 514
OPERATING SYSTEM 510
504
COMPONENT 502
NETWORK INTERFACE 540
556
562
NETWORK BUFFER 560
568
NETWORK ADAPTER 566
COMPUTER NETWORK 550

FIG. 5

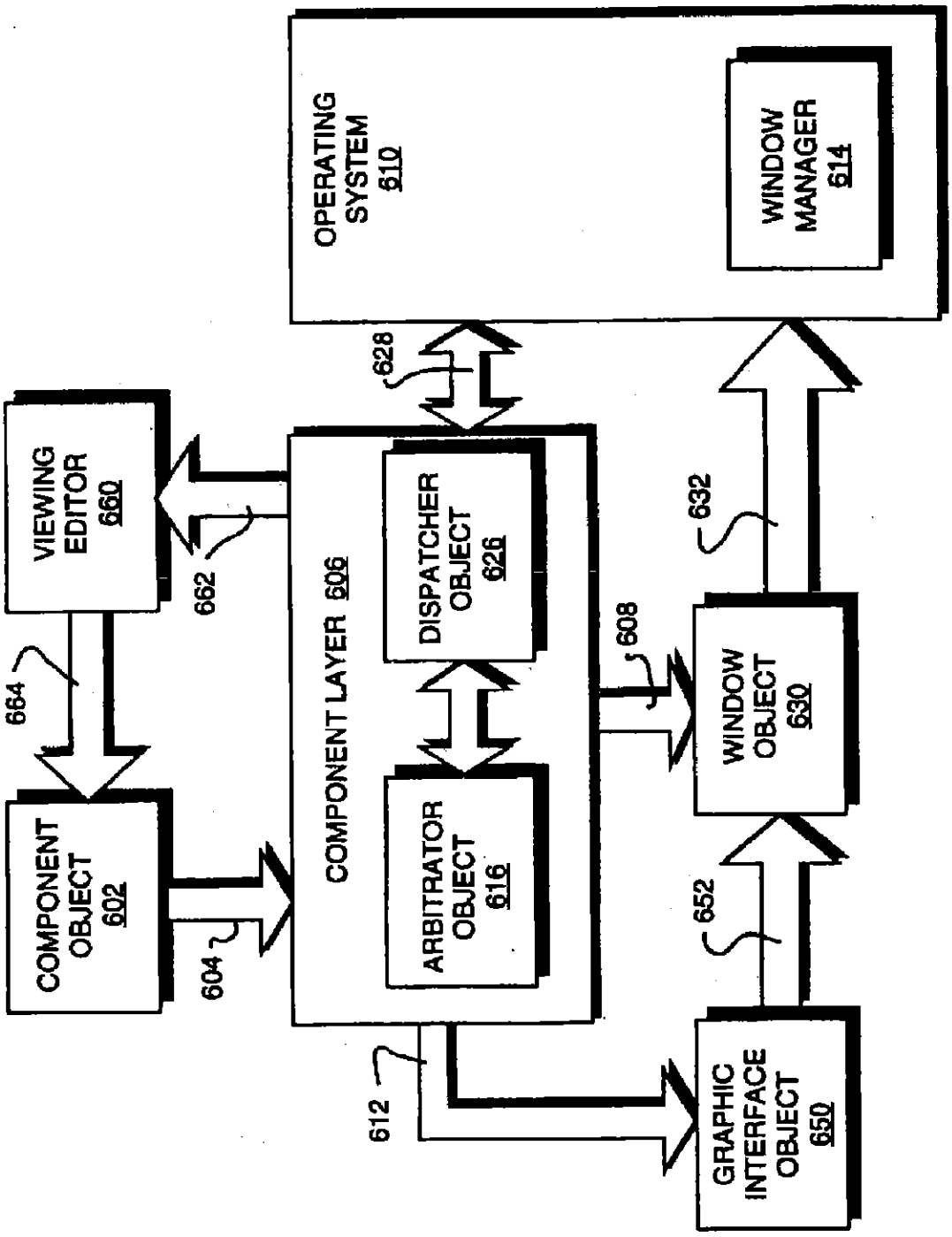

OPERATING SYSTEM 610
WINDOW MANAGER 614
628
VIEWING EDITOR 660
662
COMPONENT LAYER 606
DISPATCHER OBJECT 626
632
608
WINDOW OBJECT 630
664
COMPONENT OBJECT 602
604
ARBITRATOR OBJECT 616
652
612
GRAPHIC INTERFACE OBJECT 650

FIG. 6

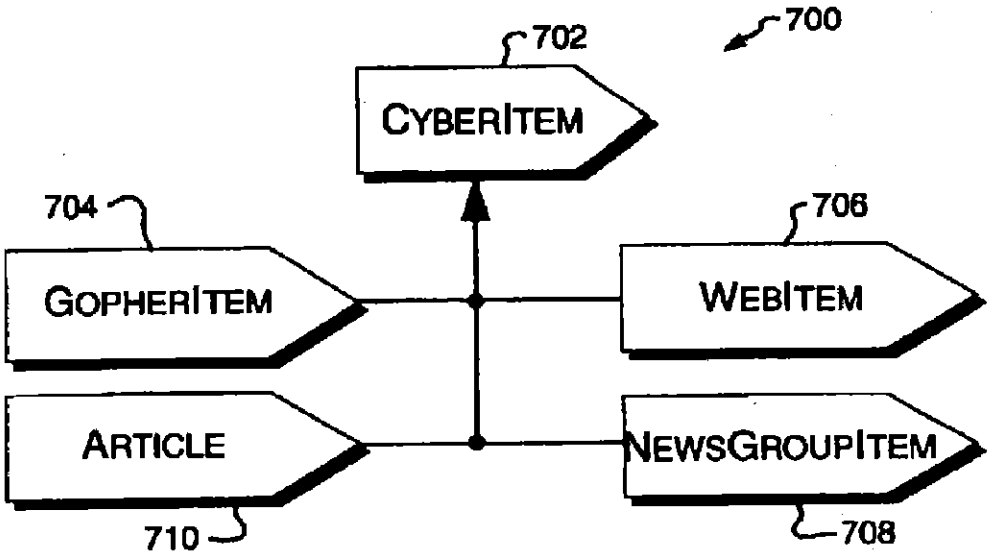

700
702
CYBERITEM
704
GOPHERITEM
706
WEBITEM
ARTICLE
NEWSGROUPITEM
710
708

FIG. 7

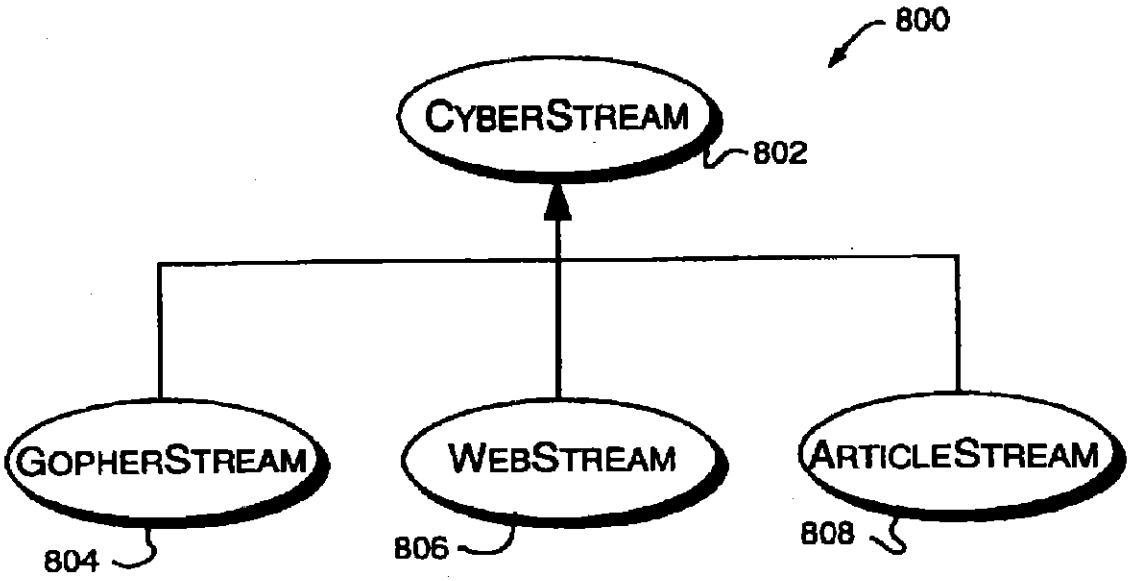

800
CYBERSTREAM
802
GOPHERSTREAM
WEBSTREAM
ARTICLESTREAM
804
806
808

FIG. 8

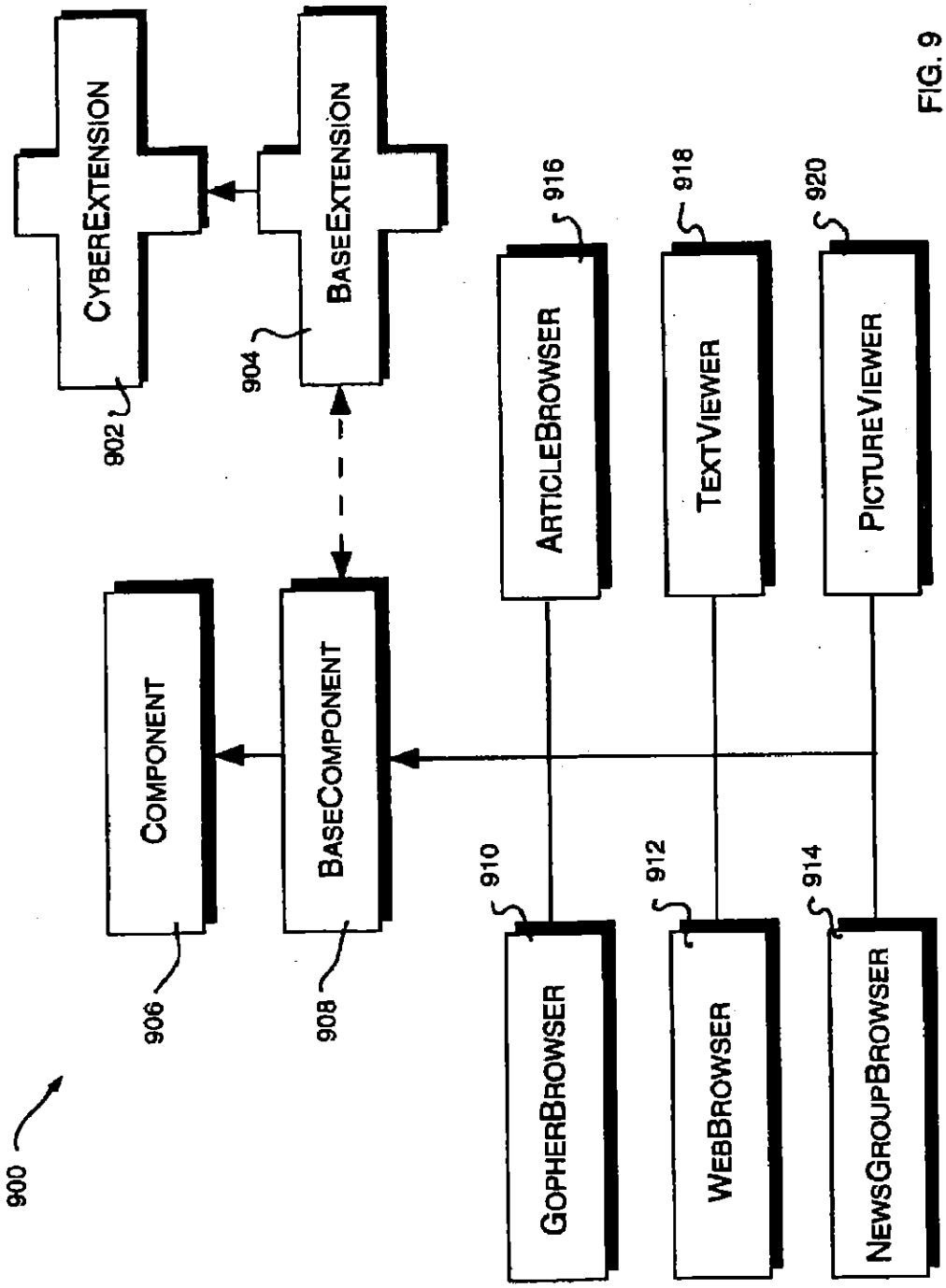

CyberExtension
BaseExtension
902
904
ArticleBrowser
TextViewer
PictureViewer
916
918
920
Component
BaseComponent
906
908
GopherBrowser
WebBrowser
NewsGroupBrowser
910
912
914
900

FIG. 9

EXTENSIBLE, REPLACEABLE NETWORK COMPONENT SYSTEM

Matter enclosed in heavy brackets [ ] appears in the original patent but forms no part of this reissue specification; matter printed in italics indicates the additions made by reissue.

CROSS REFERENCE TO RELATED APPLICATIONS

This invention is related to the following copending U.S. Patent Applications:

U.S. patent application Ser. No. 08/435,374, titled REPLACEABLE AND EXTENSIBLE NOTEBOOK COMPONENT OF A NETWORK COMPONENT SYSTEM.

U.S. patent application Ser. No. 08/435,862, titled REPLACEABLE AND EXTENSIBLE LOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,213, titled REPLACEABLE AND EXTENSIBLE CONNECTION DIALOG COMPONENT OF A NETWORK COMPONENT SYSTEM;

U.S. patent application Ser. No. 08/435,671, titled EMBEDDING INTERNET BROWSER/BUTTONS WITHIN COMPONENTS OF A NETWORK COMPONENT SYSTEM; and

U. S. patent application Ser. No. 08/435,880, tilted ENCAPSULATED NETWORK ENTITY REFERENCE OF A NETWORK COMPONENT SYSTEM, each of which was filed on May 5, 1995 and assigned to the assignee of the present invention.

FIELD OF THE INVENTION

This invention relates generally to computer networks and, more particularly, to an architecture for building Internet-specific services.

BACKGROUND OF THE INVENTION

The Internet is a system of geographically distributed computer networks interconnected by computers executing networking protocols that allow users to interact and share information over the networks. Because of such wide-spread information sharing, the Internet has generally evolved into an "open" system for which developers can design software for performing specialized operations, or services, essentially without restriction. These services are typically implemented in accordance with a client/server architecture, wherein the clients, e.g., personal computers or workstations, are responsible for interacting with the users and the servers are computers configured to perform the services as directed by the clients.

Not surprisingly, each of the services available over the Internet is generally defined by its own networking protocol. A protocol is a set of rules governing the format and meaning of messages or "packets" exchanged over the networks. By implementing services in accordance with the protocols, computers cooperate to perform various operations, or similar operations in various ways, for users wishing to "interact" with the networks. The services typically range from browsing or searching the information-having a particular data format using a particular protocol to actually acquiring information of a different format in accordance with a different protocol.

For example, the file transfer protocol (FTP) service facilitates the transfer and sharing of files across the Internet.

The Telnet service allows users to log onto computers coupled to the networks, while the network protocol provides a bulletin-board service to its subscribers. Furthermore, the various data formats of the information available on the Internet include JPEG images, MPEG movies and μ-law sound files.

Coincided with the design of these services has been the development of applications for implementing the services on the client/server architecture. Accordingly, applications are available for users to obtain files from computers connected to the Internet using the FTP protocol. Similarly, individual applications allow users to log into remote computers (as though they were logging in from terminals attached to those computers) using the Telnet protocol and, further, to view JPEG images and MPEG movies. As a result, there exists a proliferation of applications directed to user activity on the Internet.

A problem with this vast collection of application-specific protocols is that these applications are generally unorganized, thus requiring users to plod through them in order to satisfyingly, and profitably, utilize the Internet. Such lack of uniformity is time consuming and disorienting to users that want to access particular types of information but are forced to use unfamiliar applications. Because of the enormous amount of different types of information available on the Internet and the variety of applications needed to access those information types, the experience of using the Internet may be burdensome to these users.

An alternative to the assortment of open applications for accessing information on the Internet is a "closed" application system, such as Prodigy, CompuServe or America Online. Each of these systems provide a fill range of well-organized services to their subscribers; however, they also impose restrictions on the services developers can offer for their systems. Such constraint of "new" service development may be an unreasonable alternative for many users.

Two fashionable services for accessing information over the Internet are Gopher and the World-Wide Web ("Web"). Gopher consists of a series of Internet servers that provide a "list-oriented" interface to information available on the networks, the information is displayed as menu items in a hierarchical manner. Included in the hierarchy of menus are documents, which can be displayed or saved, and searchable indexes, which allow users to type keywords and perform searches.

Some of the menu items displayed by Gopher are links to information available on other servers located on the networks. In this case, the user is presented with a list of available information documents that can be opened. The opened documents may display additional lists or they may contain various data-types, such as pictures or text, occasionally, the opened documents may "transport" the user to another computer on the Internet.

The other popular information services on the Internet is the Web. Instead of providing a user with a hierarchical list-oriented view of information, the Web provides the user with a "linked-hypertext" view. Metaphorically, the Web perceives the Internet as a vast book of pages, each of which may contain pictures, text, sound, movies or various other types of data in the form of documents. Web documents are written in HyperText Markup Language (HTML) and Web servers transfer HTML documents to each other through the HyperText Transfer Protocol (HTTP).

The Web service is essentially a means for naming sources of information on the Internet. Armed with such a general naming convention that spans the entire network

system, developers are able to build information servers that potentially any user can access. Accordingly, Gopher servers, HTTP servers, FTP servers, and E-mail servers have been developed for the Web. Moreover, the naming convention enables users to identify resources (such as directories and documents) on any of these servers connected to the Internet and allow access to those resources.

As an example, a user "traverses" the Web by following hot items of a page displayed on a graphical Web browser. These hot items are hypertext links whose presence are indicated on the page by visual cues, e.g., underlined words, icons or buttons. When a user follows a link (usually by clicking on the cue with a mouse), the browser displays the target pointed to by the link which, in some cases, may be another HTML document.

The Gopher and Web information services represent entirely different approaches to interacting with information on the Internet. One follows a list-approach to information that "looks" like a telephone directory service, while the other assumes a page-approach analogous to a tabloid newspaper. However, both of these approaches include applications for enabling users to browse information available on Internet servers. Additionally, each of these applications has a unique way of viewing and accessing the information on the servers.

Netscape Navigator™ ("Netscape")is an example of a monolithic Web browser application that is configured to interact with many of the previously-described protocols, including HTFF, Gopher and FTP. When instructed to invoke an application that uses one of these protocols, Netscape "translates" the protocol to hypertext. This translation places the user farther away from the protocol designed to run the application and, in some cases, actually thwarts the user's Internet experience. For example, a discussion system requiring an interactive exchange between participants may be bogged down by hypertext translations.

The Gopher and Web services may further require additional applications to perform specific functions, such as playing sound or viewing movies, with respect to the data types contained in the documents. For example, Netscape employs helper applications for executing applications having data formats it does not "understand". Execution of these functions on a computer requires interruption of processing and context switching (i.e., saving of state) prior to invoking the appropriate applications. Thus, if a user operating within the Netscape application "opens" an MPEG movie, that browsing application number must be saved (e.g., to disk) prior to opening an appropriate MPEG application, e.g., Sparkle, to view the image. Such an arrangement is inefficient and rather disruptive to processing operations of the computer.

Typically, a computer includes an operating system and application software which, collectively, control the operations of the computer. The applications are preferably task-specific and independent, e.g., a word processor application edits text, a drawing application edits drawings and a database application interacts with information stored on a database storage unit. Although a user can move data from one application to the other, such as by copying a drawing into a word processing file, the independent applications must be invoked to thereafter manipulate that data.

Generally, the application program presents information to a user through a window of a graphical user interface by drawing images, graphics or text within the window region. The user, in turn, communicates with the application by "pointing" at graphical objects in the window with a pointer

that is controlled by a hand-operated pointing device, such as a mouse, or by pressing keys of a keyboard.

The graphical objects typically included with each window region are sizing boxes, buttons and scroll bars. These objects represent user interface elements that the user can point at with the pointer (or a cursor) to select or manipulate. For example, the user may manipulate these elements to move the windows around on the display screen, and change their sizes and appearances so as to arrange the window in a convenient manner. When the elements are selected or manipulated, the underlying application program is informed, via the window environment, that control has been appropriated by the user.

A menu bar is a further example of a user interface element that provides a list of menus available to a user. Each menu, in turn, provides a list of command options that can be selected merely by pointing to them with the mouse-controlled pointer. That is, the commands may be issued by actuating the mouse to move the pointer onto or near the command selection, and pressing and quickly releasing, i.e., "clicking" a button on the mouse.

In contrast to this typical application-based computing environment, a software component architecture provides a modular document-based computing arrangement using tools such as viewing editors. The key to document-based computing is the compound document, i.e., a document composed of many different types of data sharing the same file. The types of data contained in a compound document may range from text, tables and graphics to video and sound. Several editors, each designed to handle a particular data type of format, can work on the contents of the document at the same time, unlike the application-based computing environment.

Since many editors may work together on the same document, the compound document is apportioned into individual modules of context for manipulation by the editors. The compound-nature of the document is realized by embedding these modules within each other to create a document having a mixture of data types. The software component architecture provides the foundation for assembling documents of differing contents and the present invention is directed to a system for extending this capability to network-oriented services.

Therefore, it is among the objects of the present invention to simplify a user's experience on computer networks without sacrificing the flexibility afforded the user by employing existing protocols and data types available on those networks.

Another object of the invention is to provide a system for users to search and access information on the Internet without extensive understanding or knowledge of the underlying protocols and data formats needed to access that information.

Still another object of the invention is to provide a document-based computing system that enables users to develop modules for services directed to information available on computer networks.

Still yet another object of the invention is to provide a platform that allows third-party developers to extend a layered network component system by building new components that seamlessly interact with the system components.

SUMMARY OF THE INVENTION

Briefly, the invention comprises an extensible and replaceable network-oriented component system that pro-

vides a platform for developing network navigation components that operate on a variety of hardware and software computer system. These navigation components include key integrating components along with components, such as Gopher-specific and Web-specific components, configured to deliver conventional services directed to computer networks. Communication among these components is achieved through novel application programming interfaces (APIs) to facilitate integration with an underlying software component architecture. Such a highly-modular cooperating layered-arrangement between the network component system and the component architecture allows any existing component to be replaced, and allows new components to be added, without affecting operation of the novel network component system.

According to one aspect of the present invention, the novel system provides a network navigating service for browsing and accessing information available on the computer networks. The information may include various data types available from a variety of sources coupled to the computer networks. Upon accessing the desired information, component viewing editors are provided to modify or display, either visually or acoustically, the contents of the data types regardless of the source of the underlying data. Additional components and component viewing editors may be created in connection with the underlying software component architecture to allow integration of different data types and protocols needed to interact with information on the Internet.

In accordance with another aspect of the invention, the component system is preferably embodied as a customized framework having a set of interconnected abstract classes for defining network-oriented objects. These abstract classes include CyberItem, CyberStream and CyberExtension, and the objects they define are used to build the novel navigation components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

Specifically, CyberItem is an object abstraction which represents a "resource on a computer-network", but which may be further expanded to include resources available at any accessible location. CyberStream is an object abstraction representing a method for downloading information from a remote location on the computer network, while CyberExtension represents additional behaviors provided to the existing components for integration with the network component system.

The novel network system captures the essence of a "component-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Such a component-based system has a number of advantages. First, if a user does not like the way a particular component operates, that component can be replaced with a different component provided by another developer. In contrast, if a user does not like the way a monolithic application handles certain protocols, the only resource is to use another service because the user cannot modify the application to perform the protocol function in a different manner. Clearly, the replaceability feature of the novel network component system provides a flexible alternative to the user.

Second, the use of components is substantially less disruptive than using helper applications in situations where a monolithic application confronts differing data types and formats. Instead of "switching" application layers, the novel network system merely invokes the appropriate component and component viewing editor configured to operate with the data type and format. Such "seamless" integration among components is a significant feature of the modular cooperating architecture described herein.

A third advantage of the novel network system is directed to the cooperating relationship between the system and the underlying software computer architecture. Specifically, the novel network components are based on the component architecture technology to therefore ensure cooperation between all components in an integrated manner. The software component architecture is configured to operate on a plurality of computers, and is preferably implemented as a software layer adjoining the operating system.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of the invention may be better understood by referring to the following description in conjunction with the accompanying drawings in which:

FIG. **1** is a block diagram of a network system including a collection of computer networks interconnected by client and server computers;

FIG. **2** is a block diagram of a client component, such as a personal computer, on which the invention may advantageously operate;

FIG. **3** is a block diagram of a server computer of FIG. **1**;

FIG. **4** is a highly schematized block diagram of a layered component computing arrangement in accordance with the invention;

FIG. **5** is a schematic illustration of the interaction of a component, a software component layer and an operating system of the computer of FIG. **2**;

FIG. **6** is a schematic illustration of the interaction between a component, a component layer and a window manager in accordance with the invention;

FIG. **7** is a simplified class hierarchy diagram illustrating a base class CyberItem, and its associated subclasses, used to construct network component objects in accordance with the invention;

FIG. **8** is a simplified class hierarchy diagram illustrating a base class CyberStream, and its associated subclasses, in accordance with the invention; and

FIG. **9** is a simplified class hierarchy diagram illustrating a base class CyberExtension, and its associated subclasses, in accordance with the present invention.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

FIG. **1** is a block diagram of a network system **100** comprising a collection of computer networks **110** interconnected by client computers ("clients") **200**, e.g., workstations or personal computers, and server computers ("servers") **300**. The servers are typically computers having hardware and software elements that provide resources or services for use by the clients **200** to increase the efficiency of their operations. It will be understood to those skilled in the art that, in an alternate embodiment, the client and server may exist on the same computer; however, for the illustrative embodiment described herein, the client and server are separate computers.

Several types of computer networks **110**, including local area networks (LANs) and wide area networks (WANs), may be employed in the system **100**. A LAN is a limited area network that typically consists of a transmission medium,

such as coaxial cable or twisted pair, while a WAN may be a public or private telecommunications facility that interconnects computers widely dispersed. In the illustrative embodiment, the network system **100** is the Internet system of geographically distributed computer networks.

Computers coupled to the Internet typically communicate by exchanging discrete packets of information according to predefined networking protocols. Execution of these networking protocols allow users to interact and share information across the networks. As an illustration, in response to a user's request for a particular service, the client **200** sends an appropriate information packet to the server **300**, which performs the service and returns a result back to the client **200**.

FIG. **2** illustrates a typical hardware configuration of a client **200** comprising a central processing unit (CPU) **210** coupled between a memory **214** and input/output (I/O) circuitry **218** by bidirectional buses **212** and **216**. The memory **214** typically comprises random access memory (RAM) for temporary storage of information and read only memory (ROM) for permanent storage of the computer's configuration and basic operating commands, such as portions of an operating system (not shown). As described further herein, the operating system controls the operations of the CPU **210** and client computer **200**.

The I/O circuitry **218**, in turn, connects the computer to computer networks, such as the Internet computer networks **250**, via a bidirectional bus **222** and to cursor/pointer control devices, such as keyboard **224** (via cable **226**) and a mouse **230** (via cable **228**). The mouse **230** typically contains at least one button **234** operated by a user of the computer. A conventional display monitor **232** having a display screen **235** is also connected to I/O circuitry **218** via cable **238**. A pointer (cursor) **240** is displayed on windows **244** of the screen **235** and its position is controllable via the mouse **230** or the keyboard **224**, as is well-known. Typically, the I/O circuitry **218** receives information, such as control and data signals, from the mouse **230** and keyboard **224**, and provides that information to the CPU **210** for display on the screen **235** or, as described further herein, for transfer over the Internet **250**.

FIG. **3** illustrates a typical hardware configuration of a server **300** of the network system **100**. The server **300** has many of the same units as employed in the client **200**, including a CPU **310**, a memory **314**, and I/O circuitry **318**, each of which are interconnected by bidirectional buses **312** and **316**. Also, the I/O circuitry connects the computer to computer networks **350** via a bidirectional bus **322**. These units are configured to perform functions similar to those provided by their corresponding units in the computer **200**. In addition, the server typically includes a mass storage unit **320**, such as a disk drive, connected to the I/O circuitry **318** via bidirectional bus **324**.

It is to be understood that the I/O circuits within the computers **200** and **300** contain the necessary hardware, e.g., buffers and adapters, needed to interface with the control devices, the display monitor, the mass storage unit and the networks. Moreover, the operating system includes the necessary software drivers to control, e.g., network adapters within the I/O circuits when performing I/O operations, such as the transfer of data packets between the client **200** and server **300**.

The computers are preferably personal computers of the Macintosh® series of computers sold by Apple Computer Inc., although the invention may also be practiced in the context of other types of computers, including the IBM®G) series of computers sold by International Business Machines Corp. These computers have resident thereon, and are controlled and coordinated by, operating system software, such as the Apple® System 7®, IBM OS2®, or the Microsoft® Windows® operating systems.

As noted, the present invention is based on a modular document computing arrangement as provided by an underlying software components architecture, rather than the typical application-based environment of prior computing systems. FIG. **4** is a highly schematized diagram of the hardware and software elements of a layered component computing arrangement **400** that includes the novel network-oriented component system of the invention. At the lowest level there is the computer hardware, shown as layer **410**. Interfacing with the hardware is a conventional operating system layer **420** that includes a window manager, a graphic system, a file system and network-specific interfacing, such as a TCP/IP protocol stack and an Appletalk protocol stack.

The software component architecture is preferably implemented as a component architecture layer **430**. Although it is shown as overlaying the operating system **420**, the component architecture layer **430** is actually independent of the operating system and, more precisely, resides side-by-side with the operating system. This relationship allows the component architecture to exist on multiple platforms that employ different operating systems.

In accordance with the present invention, a novel network:oriented component layer **450** contains the underlying technology for implementing the extensible and replaceable network component system that delivers services and facilitates development of navigation components directed to computer networks, such as the Internet. As described further herein, this technology includes novel application programming interfaces (APIs) that facilitate communication among components to ensure integration with the underlying component architecture layer **430**. These novel APIs are preferably delivered in the form of objects in a class hierarchy.

It should be noted that the network component layer **450** may operate with any existing system-wide component architecture, such as the Object Linking and Embedding (OLE) architecture developed by the Microsoft Corporation; however, in the illustrative embodiment, the component architecture is preferably OpenDoc, the vendor-neutral, open standard for compound documents developed by, among others, Apple Computer, Inc.

Using tools such as viewing editors, the component architecture layer **430** creates a compound document composed of data having different types and formats. Each differing data type and format is contained in a fundamental unit called a computing part or, more generally, a "component" **460** comprised of a viewing editor along with the data content. An example of the computing component **460** may include a MacDraw component. The editor, on the other hand, is analogous to an application program in a conventional computer. That is, the editor is a software component which provides the necessary functionality to display a component's contents and, where appropriate, present a user interface for modifying those contents. Additionally, the editor may include menus, controls and other user interface elements.

According to the invention, the network component layer **450** extends the functionality of the underlying component architecture layer **430** by defining network-oriented components **480**. Included among these components are key inte-

grating components (such as notebook, log and connection dialog components) along with components configured to deliver conventional services directed to computer networks, such as Gopher-specific and Web-specific components. Moreover, the components may include FTP-specific components for transferring files across the networks. Telnet-specific components for remotely logging onto other computers, and JPEG-specific and MPEG-specific components for viewing image and movie data types and formats.

A feature of the invention is the ability to easily extend, or replace, any of the components of the layered computing arrangement **400** with a different component to provide a user with customized network-related services. As described herein, this feature is made possible by the cooperating relationship between the network component layer **450** and its underlying component architecture layer **430**. The integrating components communicate and interact with these various components of the system in a "seamlessly integrated" manner to provide basic tools for navigating the Internet computer networks.

FIG. **4** also illustrates the relationship of applications **490** to the elements of the layered computing arrangement **400**. Although they reside in the same "user space" as the components **460** and network components **480**, the applications **490** do not interact with these elements and, thus, interface directly to the operating system layer **420**. Because they are designed as monolithic, autonomous modules, applications (such as previous Internet browsers) often do not even interact among themselves. In contrast, the components of the arrangement **400** are designed to work together via the common component architecture layer **430** or, in the case of the network components, via the novel network component layer **450**.

Specifically, the invention features the provision of the extensible and replaceable network-oriented component system which, when invoked, causes actions to take place that enhance the ability of a user to interact with the computer to search for, and obtain, information available over computer networks such as the Internet. The information is manifested to a user via a window environment, such as the graphical user interface provide by System 7 or Windows, that is preferably displayed on the screen **235** (FIG. **2**) as a graphical display to facilitate interactions between the user and the computer, such as the client **200**. This behavior of the system is brought about by the interaction of the network components with a series of system software routines associated with the operating system **420**. These system routines, in turn, interact with the components architecture layer **430** to create the windows and graphical user interface elements, as described further herein.

The window environment is generally part of the operating system software **420** that includes a collection of utility programs for controlling the operation of the computer **200**. The operating system, in turn, interacts with the components to provide higher levels functionality, including a direct interface with the user. A component makes use of operating system functions by issuing a series of task commands to the operating system via the network component layer **450** or, as is typically the case, through the component architecture layer **430**. The operating system **420** then performs the requested task. For example, the component may request that a software driver of the operating system initiate transfer of a data packet over the networks **250** or that the operating system display certain information on a window for presentation to the user.

FIG. **5** is a schematic illustration of the interaction of a component **502**, software component layer **506** and an operating system **510** of a computer **500**, which is similar to, and has equivalent elements of, the client computer **200** of FIG. **2**. As noted, the network component layer **450** (FIG. **4**) is integrated with the computer architecture layer **430** to provide a cooperating architecture that allows any component to be replaced or extended, and allows new components to be added, without affecting operation of the network component system, accordingly, for purposes of the present discussion, the layers **430** and **450** may be treated as a single software component layer **506**.

The component **502**, component layer **506** and operating system **510** interact to control and coordinate the operations of the computer **500** and their interaction is illustrated schematically by arrows **504** and **508**. In order to display information on a screen display **535**, the component **502** and component layer **506** cooperate to generate and send display commands to a window manager **514** of the operating system **510**. The window manager **514** stores information directly (via arrow **516**) into a screen buffer **520**.

The window manager **514** is a system software routine that is generally responsible for managing windows **544** that the user views during operation of the network component system. That is, it is generally the task of the window manager to keep track of the location and size of the window and window areas which must be drawn and redrawn in connection with the network component system of the present invention.

Under control of various hardware and software in the system, the contents of the screen buffer **520** are read out of the buffer and provided, as indicated schematically by arrow **522**, to a display adapter **526**. The display adapter contains hardware and software (sometimes in the form of firmware) which converts the information in the screen buffer **520** to a form which can be used to drive a display screen **535** of a monitor **532**. The monitor **532** is connected to display adapter **526** by cable **528**.

Similarly, in order to transfer information as a packet over the computer networks, the component **502** and component layer **506** cooperate to generate and send network commands, such as remote procedure calls, to a network-specific interface **540** of the operating system **510**. The network interface comprises system software routines, such as "stub" procedure software and protocol stacks, that are generally responsible for forming the information into a predetermined packet format according to the specific network protocol used, e.g., TCP/IP or Apple-talk protocol.

Specifically, the network interface **540** stores the packet directly (via arrow **556**) into a network buffer **560**. Under control of the hardware and software in the system, the contents of the network buffer **560** are provided, as indicated schematically by arrow **562**, to a network adapter **566**. The network adapter incorporates the software and hardware, i.e., electrical and mechanical interchange circuits and characteristics, needed to interface with the particular computer networks **550**. The adapter **566** is connected to the computer networks **550** by cable **568**.

In a preferred embodiment, the invention described herein is implemented in an object-oriented programming (OOP) language, such as C++, using System Object Model (SOM) technology and OOP techniques. The C++ and SOM languages are well-known and many articles and texts are available which describe the languages in detail. In addition, C++ and SOM compilers are commercially available from several vendors. Accordingly, for reasons of brevity, the details of the C++ and SOM languages and the operations of their compilers will not be discussed further in detail herein.

As will be understood by those skilled in the art, OOP techniques involve the definition, creation, use and destruction of "objects". These objects are software entities comprising data elements and routines, or functions, which manipulate the data elements. The data and related functions are treated by the software as an entity that can be created, used and deleted as if it were a single item. Together, the data and functions enable objects to model virtually any real-world entity in terms of its characteristics, which can be represented by the data elements, and its behavior, which can be represented by its data manipulation functions. In this way, objects can model concrete things like computers, while also modeling abstract concepts like numbers or geometrical designs.

Objects are defined by creating "classes" which are not objects themselves, but which act as templates that instruct the compiler how to construct an actual object. A class may, for example, specify the number and type of data variables and the steps involved in the functions which manipulate the data. An object is actually created in the program by means of a special function called a "constructor" which uses the corresponding class definition and additional information, such as arguments provided during object creation, to construct the object. Likewise objects are destroyed by a special function called a "destructor". Objects may be used by manipulating their data and invoking their functions.

The principle benefits of OOP techniques arise out of three basic principles encapsulation, polymorphism and inheritance. Specifically, objects can be designed to hide, or encapsulate all, or a portion of, its internal data structure and internal functions. More specifically, during program design, a program developer can define objects in which all or some of the data variables and all or some of the related functions are considered "private" or for use only by the object itself. Other data or functions can be declared "public" or available for use by other programs. Access to the private variables by other programs can be controlled by defining public functions for an object which access the object's private data. The public functions form a controlled and consistent interface between the private data and the "outside" world. Any attempt to write program code which directly accesses the private variables causes the compiler to generate an error during program compilation which error stops the compilation process and prevents the program from being run.

Polymorphism is a concept which allows objects and functions that have the same overall format, but that work with different data, to function differently in order to produce consistent results. Inheritance, on the other hand, allows program developers to easily reuse pre-existing programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to declare classes (and the objects which are later created from them) as related. Specifically, classes may be designated as subclasses of other base classes. A subclass "inherits" and has access to all of the public functions of its base classes just as if these functions appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions or may modify some or all of its inherited functions merely by defining a new function with the same form (overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass). The creation of a new subclass which has some of the functionality (with selective modification) of another class allows software developers to easily customize existing code to meet their particular needs.

In accordance with the present invention, the component **502** and windows **544** are "objects" created by the component layer **506** and the window manager **514**, respectively, the latter of which may be an object-oriented program. Interaction between a component, component layer and a window manager is illustrative in greater detail in FIG. **6**.

In general, the component layer **606** interfaces with the window manager **614** by creating and manipulating objects. The window manger itself may be an object which is created when the operating system is started. Specifically, the component layer creates window objects **630** that create the window manager to create associated windows on the display screen. This is shown schematically by an arrow **608**. In addition, the component layer **606** creates individual graphic interface objects **650** that are stored in each window object **630**, as shown schematically by arrows **612** and **652**. Since many graphic interface objects may be created in order to display many interface elements on the display screen, the window object **630** communicates with the window manager by means of a sequence of drawing commands issued from the window object to the window manager **614**, as illustrated by arrow **632**.

As noted, the component layer **606** functions to embed components within one another to form a component document having mixed data types and formats. Many different viewing editors may work together to display, or modify, the data contents of the documents. In order to direct keystrokes and mouse events initiated by a user to the proper components and editors, the component layer **606** includes an arbitrator **616** and a dispatcher **626**.

The dispatcher is an object that communicates with the operating system **610** to identify the correct viewing editor **660**, while the arbitrator is an object that informs the dispatcher as to which editor "owns" the stream of keystrokes or mouse events. Specifically, the dispatcher **626** receives these "human-interface" events from the operating system **610** (as shown schematically by arrow **628**) and delivers them to the correct viewing editor **660** via arrow **662**. The viewing editor **660** then modifies or displays, either visually or acoustically, the contents of the data types.

Although OOP offers significant improvements over other programming concepts, software development still requires significant outlays of time and effort, especially if no pre-existing software is available for modulation. Consequently, a prior art approach has been to provide a developer with a set of preferred, interconnected classes which create a set of objects and additional miscellaneous routines that are all directed to performing commonly-encountered tasks in a particular environment. Such predefined classes and libraries are typically called "frameworks" and essentially provide a pre-fabricated structure for a working document.

For example, a framework for a user interface might provide a set of predefined graphic interface objects which create windows, scroll bars, menus, etc. and provide the support and "default" behavior for these interface objects. Since frameworks are based on object-oriented techniques, the predefined classes can be used as base classes and the built-in default behavior can be inherited by developer-defined subclasses and either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending the capabilities of that original program. In addition, developers are not blindly working through layers of code because the framework providers architectural guidance and modeling and, at the same time, frees the developers to supply specific actions unique to the problem domain.

There are many kinds of framework available, depending on the level of the system involved and the kind of problem to be solved. The types of frameworks range from high-level frameworks that assist in developing a user interface, to lower-level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application-type frameworks include MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXT Step App Kit (NeXT) and Smalltalk-80 MVC (ParcPlace).

While the framework approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over other programming techniques, there are difficulties which arise. These difficulties are caused by the fact that it is easy for developers to reuse their own objects, but it is difficult for the developers to use objects generated by other programs. Further, frameworks generally consist of one or more object "layers" on top of a monolithic operating system and even with the flexibility of the object layer, it is still often necessary to directly interact with the underlying system by means of awkward procedure calls.

In the same way that a framework provides the developer with prefab functionality for a document, a system framework, such as that included in the preferred embodiment, can provide a prefab functionality for system level services which developers can modify or override to create customized solutions, thereby avoiding the awkward procedural calls necessary with the prior art framework. For example, consider a customizable network interface framework which can provide the foundation for browsing and accessing information over a computer network. A software developer who needed these capabilities would ordinarily have to write specific routines to provide them. To do this with a framework, the developer only needs to supply the characteristic and behavior of the finished output, while the framework provides the actual routines which perform the tasks.

A preferred embodiment takes the concept of frameworks and applies it throughout the entire system, including the document, component, component layer and the operating system. For the commercial or corporate developer, systems integrator, or OEM, this means all of the advantages that have been illustrated for a framework, such as MacApp, can be leveraged not only at the application level for things such as text and graphical user interfaces, but also at the system level for such services as printing, graphics, multi-media, file systems and, as described herein, network-specific operations.

Referring again to FIG. **6**, the window object **630** and the graphic interface object **650** are elements of a graphical user interface of a network component system having a customizable framework for greater enhancing the ability of a user to navigate or browse through information stored on servers coupled to the network. Moreover, the novel network system provides a platform for developing network navigation components for operation on a variety of hardware and software computer systems.

As noted, the network components are preferably implemented as objects and communication among the network component objects is effected through novel application programming interfaces (APIs). These APIs are preferably delivered in the form of objects in a class hierarchy that is extensible so that developers can create new components and editors. From an implementation viewpoint, the objects can be subclassed and can inherit from base classes to build customized components allow users to see different kinds of data using different kinds of protocols, or to create components that function differently from existing components.

In accordance with the invention, the customized framework has a set of interconnected abstract classes for defining network-oriented objects used to build these customized network components. These abstract classes include CyberItem, CyberStream and CyberExtension and the objects they define are used to build the novel network components. Interactions among these latter components and existing components of the underlying software architecture provide the basis for the extensibility and replaceability features of the network component system.

In order to further understand the operations of these network component objects, it may be useful to examine their construction together with the major function routines that comprise the behavior of the objects. In examining the objects, it is also useful to examine the classes that are used to construct the objects (as previously mentioned the classes serve as templates for the construction of objects). Thus, the relation of the classes and the functions inherent in each class can be used to predict the behavior of an object once it is constructed.

FIG. **7** illustrates a simplified class hierarchy diagram **700** of the base class CyberItem **702** used to construct the network component object **602**. In general, CyberItem is an abstraction that may represent resources available at any location accessible from the client **200**. However, in accordance with the illustrative embodiment, a CyberItem is preferably a small, self-contained object that represents a resource, such as a service, available on the Internet and subclasses of the CyberItem base class are used to construct various network component objects configured to provide such services for the novel network-oriented component system.

For example, the class GopherItem **704** may be used to construct a network component object representing a "thing in Gopher space", such as a Gopher directory, while the subclass WebItem **706** is derived from CyberItem and encapsulates a network component object representing a "thing in Web space, e.g. a Web page. Similarly, the subclass NewsGroupItem **708** may be used to construct a network object representing a newsgroup and the class Article **710** is configured to encapsulate a network component object representing an article resource on an Internet server.

Since each of the classes used to construct these network component objects are subclasses of the CyberItem base class, each class inherits the functional operators and methods that are available from that base class. For example, methods associated with the CyberItem base class for returning an icon family and a name are assumed by the subclasses to allow the network components to display CyberItem objects in a consistent manner. The methods associated with the CyberItem base class include (the arguments have been omitted for simplicity):

```
GetRefCount ();
IncrementRefCount ();
Release ();
SetUpFromURL ();
ExternalizeContent ();
StreamToStorageUnit ();
StreamFromStorageUnit ();
Clone ();
Compare ();
GetStringProperty ();
```

SetDefaultName ();
GetURL ();
GetIconSuite ();
CreateCyberStream ();
Open ();
OpenInFrame ();
FindWindow ().

In some instances, a CyberItem object may need to spawn a CyberStream object in order to obtain the actual data for the object it represents. FIG. **8** illustrates a simplified class hierarchy diagram **800** of the base class CyberStream **802**. As noted, CyberStream is an abstraction that serves as an API between a component configured to display a particular data format and the method for obtaining the actual data. This allows developers to design viewing editors that can display the content of data regardless of the protocol required to obtain that data.

For example, a developer may design a picture viewing editor that uses the CyberStream API to obtain data bytes describing a picture. The actual data bytes are obtained by a subclass of CyberStream configured to construct a component object that implements a particular protocol, such as Gopher and HTTP. That is, the CyberStream object contains the software commands necessary to create a "data stream" for transferring information from one object to another. According to the invention, a GopherStream subclass **804** is derived from the CyberStream base class and encapsulates a network object that implements the Gopher protocol, while the class WebStream **806** may be used to construct a network component configured to operate in accordance with the HTTP protocol.

The methods associated with the CyberStream class, and which are contained in the objects constructed from the subclasses, include (the arguments have been omitted for simplicity):

GetStreamStatus ();
GetTotalDataSize ();
GetStreamError ();
GetStatusString ();
OpenWithCallback ();
Open ();
GetBuffer ();
ReleaseBuffer ();
Abort ().

FIG. **9** is a simplified class hierarchy diagram of the base class CyberExtension **902** which represents additional behaviors provided to components of the underlying software component architecture. Specifically, CyberExtension are the mechanisms for adding functionality to, and extending the APIs of, existing components so that they may communicate with the novel network components. As a result, the CyberExtension base class **902** operates in connection with a Component base class **906** through their respective subclasses BaseExtension **904** and BaseComponent **908**.

The CyberExtension base class provides an API for accessing other network-specific components, such as notebooks and logs, and for supporting graphical user interface elements, such as menus. CyberExtension objects are used by components that display the contents of CyterItem objects. This includes browser-like components such as a Gopher browser or Web browser, as well as JPEG-specific components which display particular types of data such as pictures. The CyberExtension objects also keep track of the CyberItem objects which these components are responsible for displaying.

In accordance with the invention, the class GopherBrowser **910** may be used to construct a Gopher-like network browsing component and the class WebBrowser **912** may be able to construct a Web-like network browsing component. Likewise, a TextViewer subclass **918** may encapsulate a network component configured to display text and a PictureViewer subclass **920** may construct a component for displaying pictures. The methods associated with the CyberExtension class include (the arguments have been omitted for simplicity):

ICyberExtension ( );
Components displaying the contents of CyberItem object

SetCyberItem ( );
GetCyberItem ( );
GetCyberItemWindow ( );
IsCyberItemSelected ( );
GetSelectedCyberItems ( );
Notebook and Log Tools

AddCyberItemToLog ( );
ShowLogWindow ( );
IsLogWindowShown ( );
AddCyberItemToNotebook ( );
AddCyberItemsToNotebook ( );
ShowNotebookWindow ( );
IsNotebookWindowShown ( );
SetLogFinger ( );
ClearLogFinger ( );
Notebook and Log Menu Handlers

InstallServicesMenu ( );
AdjustMenus ( );
DoCommand ( ).

In summary, the novel network system described herein captures, the essence of a "comprehensive-based" approach to browsing and retrieving network-oriented information as opposed to the monolithic application-based approach of prior browsing systems. Advantages of such a component-based system include the ability to easily replace and extend components because of the cooperating relationship between the novel network-oriented component system and the underlying component architecture. This relationship also facilitates "seamless" integration and cooperation between components and component viewing editors when confronted with differing data types and formats.

While there has been shown and described an illustrative embodiment for implementing an extensible and replaceable network component system, it is to be understood that various other adaptations and modifications may be made within the spirit and scope of the invention. For example, additional system software routines may be used when implementing the invention in various applications. These additional system routines include dynamic link libraries (DLL), which are program files containing collections of window environment and networking functions designed to perform specific classes of operations. These functions are invoked as needed by the software component layer to perform the desired operations. Specifically, DLLs, which are generally well-known, may be used to interact with the component layer and window manager to provide network-specific components and functions.

The foregoing description has been directed to specific embodiments of this invention. It will be apparent, however, that other variations and modifications may be made to the described embodiments, with the attainment of some or all of their advantages. Therefor, it is the object of the appended claims to cover all such variations and modifications as come within the true spirit and scope of the invention.

What is claimed is:

1. An extensible and replaceable layered component computing arrangement residing on a computer coupled to a computer network, the layered arrangement comprising:

a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components; and

a network component layer for developing network navigation components that provide services directed to the computer network, the network component layer includes application programming interfaces; and

a first class included in the application programming interfaces to construct a first network navigation object that represents different network resources available on the computer network, wherein the network component layer coupled to the software component architecture layer in integrating relation to facilitate communication among the computing and network navigation components.

2. The computing arrangement of claim 1 wherein the network navigation components are objects.

3. The computing arrangement of claim 1 wherein the application programming interfaces further comprise a second class for constructing a second network navigation object representing a data stream for transferring information among objects of the arrangement.

4. The computing arrangement of claim 3 wherein the first network navigation object is an Item object and the second network navigation object is a Stream object, and wherein the Item object spawns the Stream object to obtain information from the network resource that the Item object represents.

5. The computing arrangement of claim 3 wherein the application programming interfaces further comprise a third class for constructing a third network navigation object representing additional behaviors provided to computing components of the software component architecture layer to thereby enable communication between the computing components and the network navigation components.

6. An extensible and replaceable layered component computing arrangement for providing services directed to information available on computer networks, the computing arrangement comprising:

a processor;

an operating system;

a software component architecture layer coupled to the operating system to control the operations of the processor, the software component architecture layer defining a plurality of computing components; and

a network component layer for creating network navigation components configured to search and obtain information available on the computer networks, the network component layer includes application programming interfaces; and

means for constructing a network navigation component that represents different resources available on the computer network, wherein the network component layer is integrally coupled to the software component architecture layer to ensure communication among the computing and network navigation components.

7. The computing arrangement of claim 6 wherein the network component layer and software component architecture layer comprise means for embedding components within one another to form a compound document having mixed data types and formats.

8. The computing arrangement of claim 6 wherein the application programming interfaces comprise means for constructing a network navigation component that implements a protocol.

9. The computing arrangement of claim 6 wherein the application programming interfaces comprise means for constructing a network navigation component that provides additional functionality to existing computing components to enable communication among the components.

10. The computing arrangement of claim 9 wherein the computing component comprises a computing part having a viewing editor and data content.

11. The computing arrangement of claim 10 wherein the computing component functions to one of transfer files over the networks, remotely log onto another computer coupled to the networks and view images on a screen of the computing arrangement.

12. The computing arrangement of claim 10 wherein the network navigation component comprises a browsing component.

13. The computing arrangement of claim 10 wherein the network navigation component comprises a component for one of displaying text and displaying movies on a screen of the computing arrangement.

14. An extensible and replaceable layered component computing arrangement residing on a computer adapted to be coupled on a computer network, the layered arrangement comprising:

a software component architecture layer interfacing with an operating system to control the operations of the computer, the software component architecture layer defining a plurality of computing components;

a network component layer adapted to be coupled to at least one network navigation component that provides a service directed to the computer network, the network component layer including an application programming interface; and

a number of interconnected abstract classes included in the application programming interface, at least on abstract class for defining a network navigation object that represents a resource available on the computer network, the network component layer coupled to the software component architecture layer to facilitate communication among the network navigation component and at least one computing component.

15. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents a method of downloading information from a remote location on the computer network.

16. The layered arrangement of claim 14, wherein the abstract class defines a network navigation object that represents additional behaviors provided to the computing components of the software component architecture layer for integrating with the network component layer.

17. The layered arrangement of claim 14, wherein the network navigation object is adapted to browse the computer network.

18. The layered arrangement of claim 14, wherein the network navigation object is adapted to display text on a computer display.

19. The layered arrangement of claim 14, wherein the network navigation object is adapted to display images on a computer display.

20. The layered arrangement of claim 14, wherein the network navigation object includes software commands for creating a datastream for transferring information between objects in the layered component computing arrangement.

* * * * *

EXHIBIT J

US005455854A

United States Patent [19]

Dilts et al.

[11] Patent Number: 5,455,854

[45] Date of Patent: Oct. 3, 1995

[54] **OBJECT-ORIENTED TELEPHONY SYSTEM**

[75] Inventors: **Michael R. Dilts**, Saratoga; **Steven H. Milne**, Palo Alto; **David B. Goldsmith**, Los Gatos, all of Calif.

[73] Assignee: **Taligent, Inc.**, Cupertino, Calif.

[21] Appl. No.: **108,877**

[22] Filed: **Oct. 26, 1993**

[51] **Int. Cl.**[6] **H04M 3/42**; H04M 11/00; G06F 1/00

[52] **U.S. Cl.** **379/201**; 379/94; 364/DIG. 1; 395/62; 395/159; 395/917

[58] **Field of Search** 379/93, 94, 201, 379/207; 364/200; 395/159, 160, 161, 500, 51, 75, 917, 922, 61, 62, 700

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,653,090	3/1987	Hayden	379/204
4,785,408	11/1988	Britton et al.	379/97 X
4,821,220	4/1989	Duisberg	364/578
4,885,717	12/1989	Beck et al.	364/900
4,891,630	1/1990	Friedman et al.	340/706
4,953,080	8/1990	Dysart et al.	364/200
5,041,992	8/1991	Cunningham et al.	364/518
5,050,090	9/1991	Golub et al.	364/478
5,060,276	10/1991	Morris et al.	382/8
5,075,848	12/1992	Lai et al.	395/425
5,093,914	3/1992	Coplien et al.	395/700
5,119,475	6/1992	Smith et al.	395/156
5,125,091	6/1992	Staas, Jr. et al.	395/650
5,133,075	7/1992	Risch	395/800
5,136,705	8/1992	Stubbs et al.	395/575
5,151,987	9/1992	Abraham et al.	395/375
5,181,162	1/1993	Smith et al.	364/419
5,241,588	8/1993	Babson, III et al.	379/94 X
5,323,452	6/1994	Dickman et al.	379/207 X

FOREIGN PATENT DOCUMENTS

0463207	1/1992	European Pat. Off. .
3932686	4/1990	Germany .
4131380	3/1993	Germany .

OTHER PUBLICATIONS

Fredrik Ljungblom, "A Service Management Systems for The Intelligent Network", Ericsson Review No. 1, 1990, pp. 32–41.

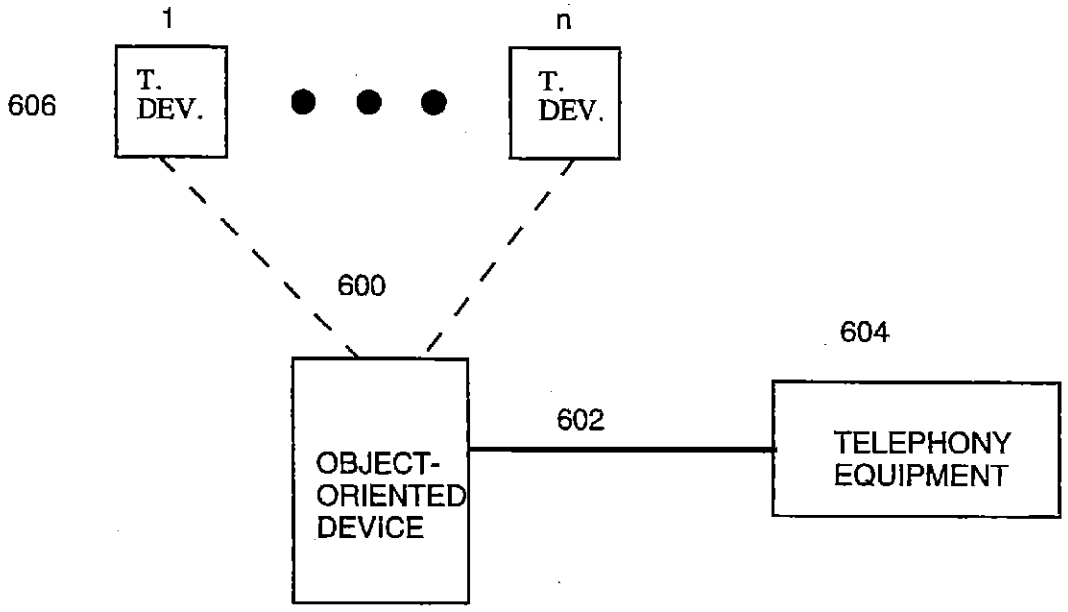
Kathleen O. Rankin, "Boost '93: dishing out healthy portions of reality", Sep., 1993, Bellcore Exchange, pp. 21–24.

Primary Examiner—Jeffery A. Hofsass
Assistant Examiner—Harry S. Hong
Attorney, Agent, or Firm—Keith Stephens

[57] **ABSTRACT**

A method and system for enabling a set of object interface application elements and telephony system elements. Particular objects may be chosen depending on which elements of the telephony system will need to be interfaced. A particular object is capable of interfacing with one or more elements of the telephony system. The elements of the telephony system may be any identifiable aspect of the telephony system. For example, the objects could represent a handset or a line. Less tangible elements can also be represented, such as signals or procedures, including call progress tones, call setup, call hold, conference calls, or other call features.

24 Claims, 21 Drawing Sheets



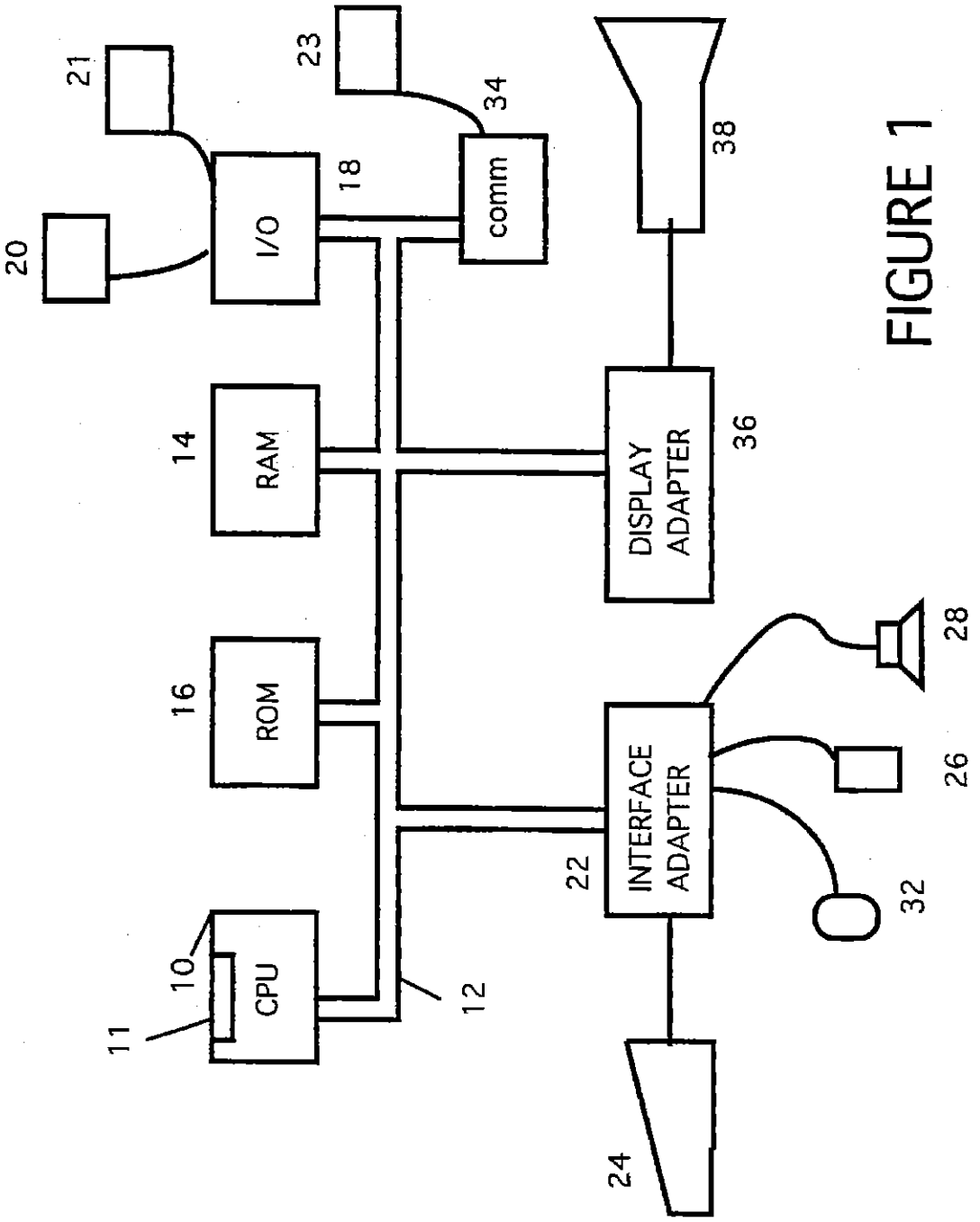

21
23
34
38
18
I/O
comm
20
14
RAM
DISPLAY ADAPTER
36
16
ROM
28
INTERFACE ADAPTER
26
22
32
10
CPU
11
12
24

FIGURE 1

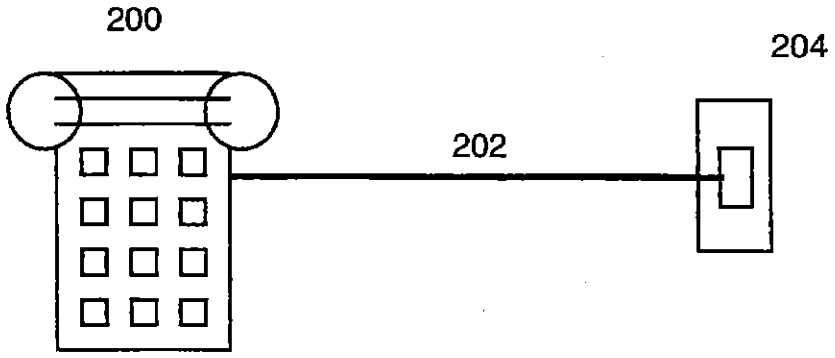


Figure 2

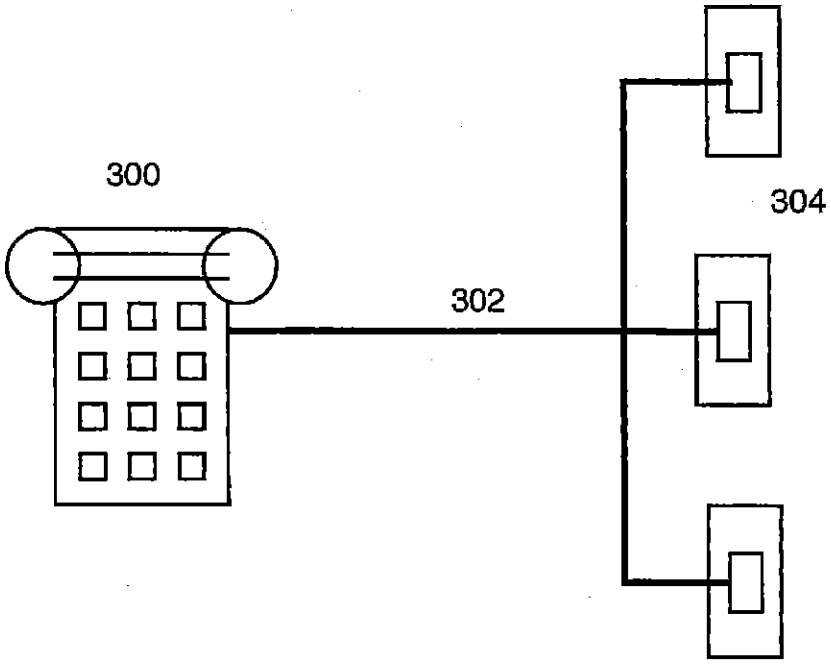


Figure 3

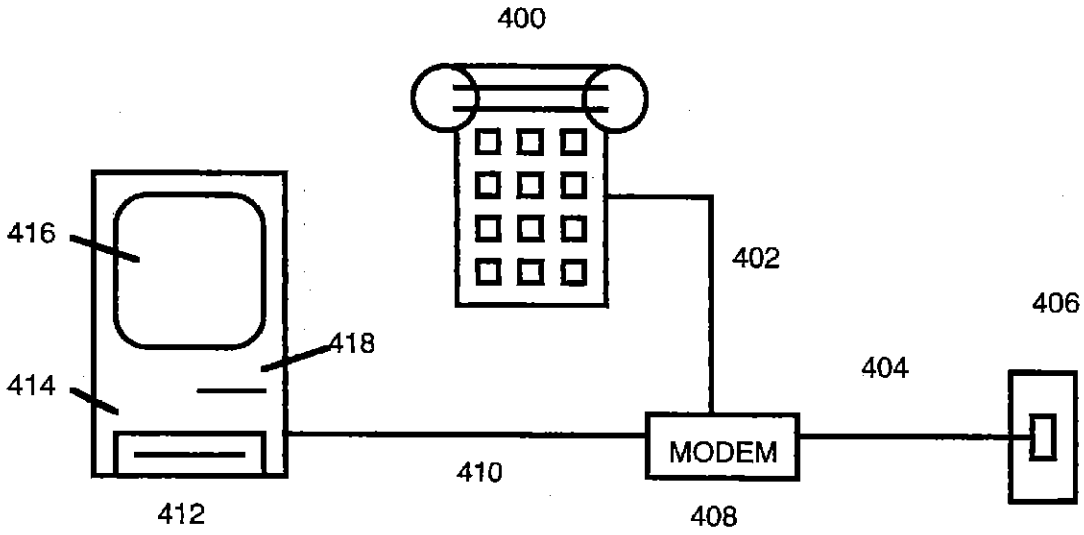


Figure 4

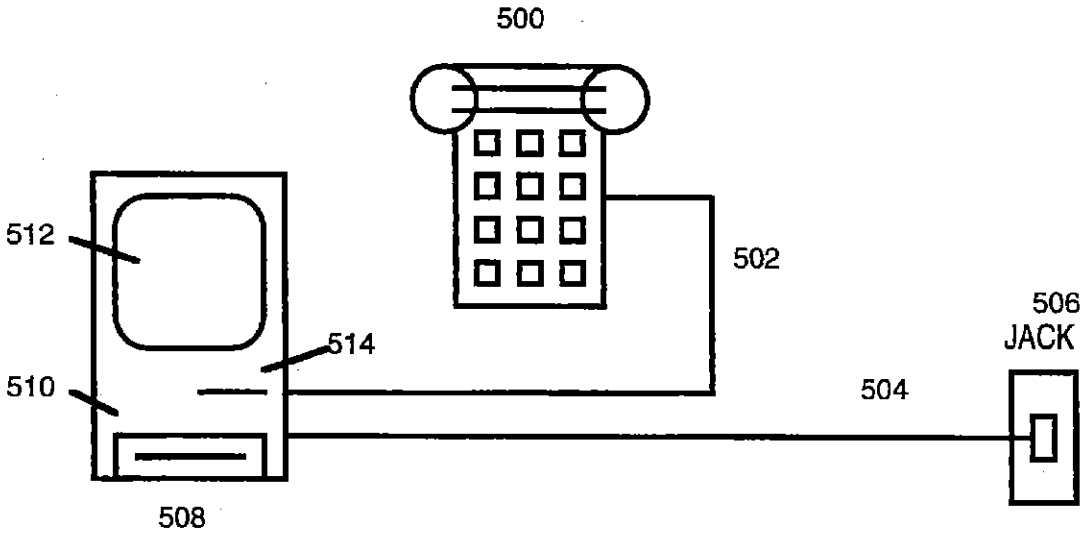


Figure 5

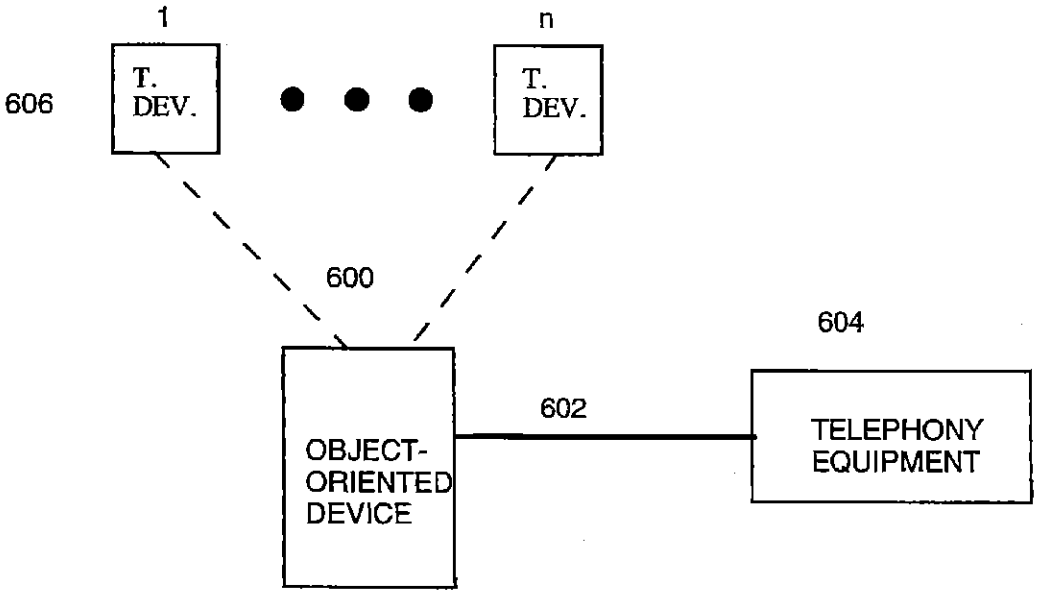


Figure 6

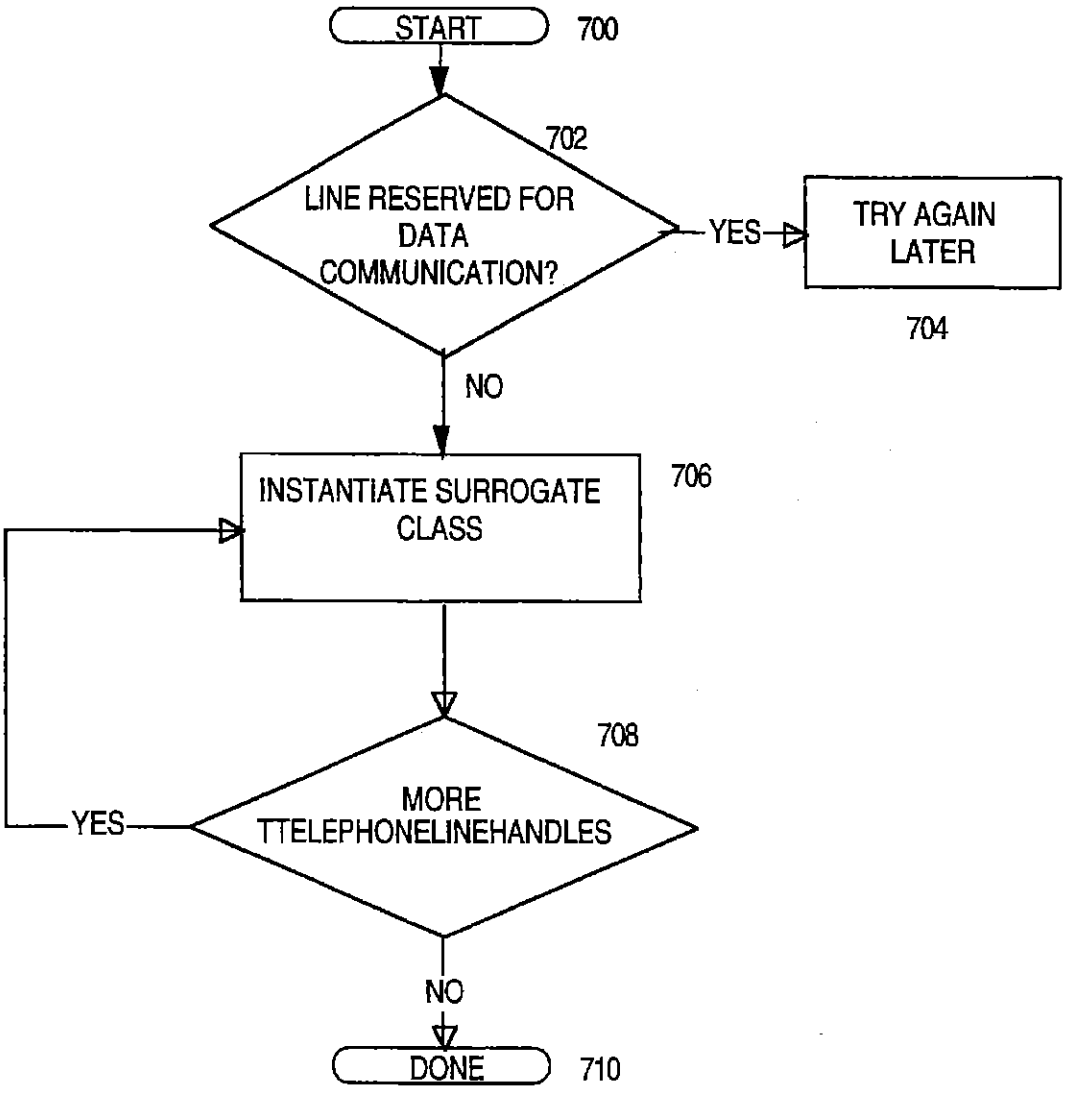

START
700
702
LINE RESERVED FOR DATA COMMUNICATION?
YES
TRY AGAIN LATER
704
NO
INSTANTIATE SURROGATE CLASS
706
708
MORE TTELEPHONELINEHANDLES
YES
NO
DONE
710

Figure 7

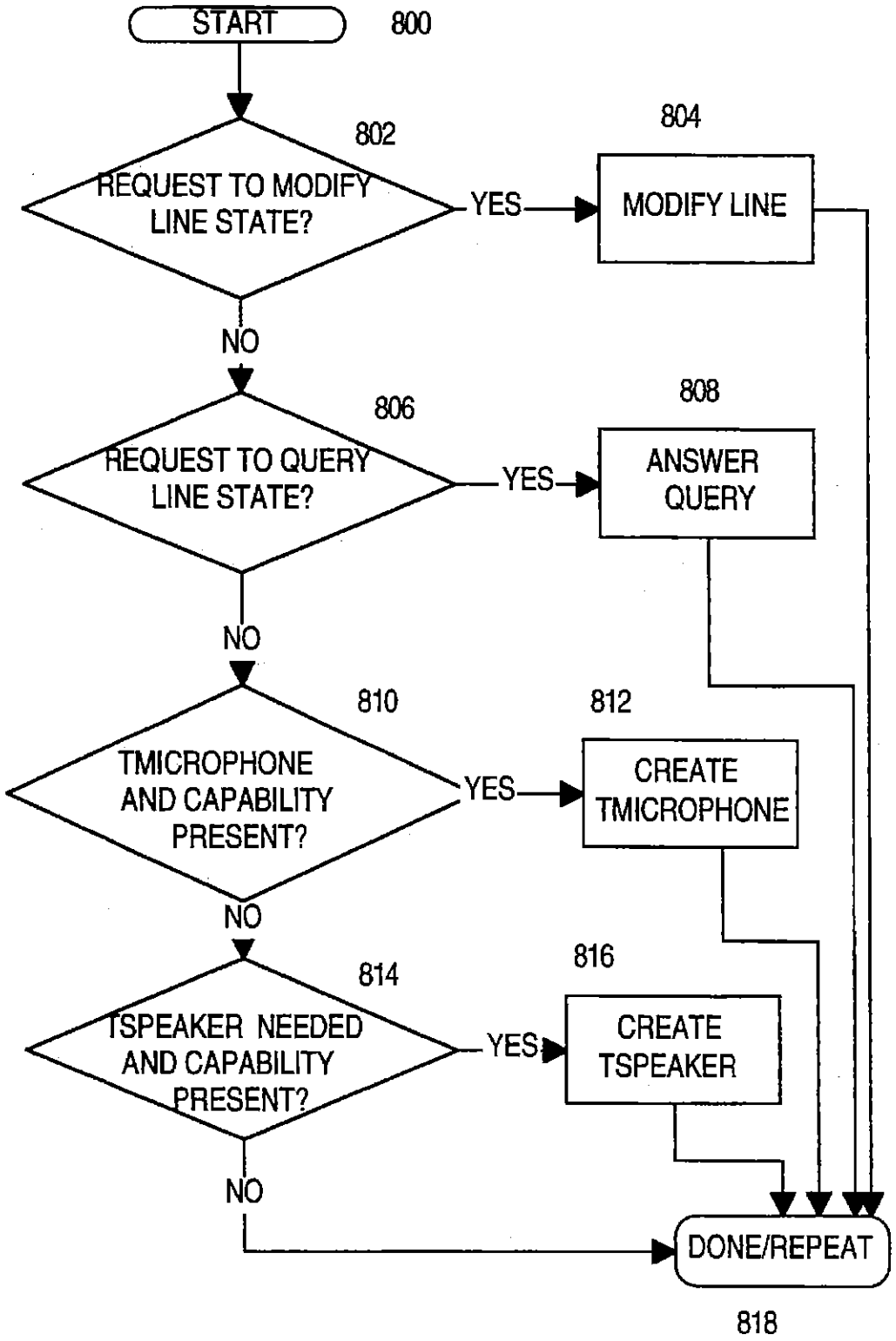

START
800
802
REQUEST TO MODIFY LINE STATE?
YES
804
MODIFY LINE
NO
806
REQUEST TO QUERY LINE STATE?
YES
808
ANSWER QUERY
NO
810
TMICROPHONE AND CAPABILITY PRESENT?
YES
812
CREATE TMICROPHONE
NO
814
TSPEAKER NEEDED AND CAPABILITY PRESENT?
YES
816
CREATE TSPEAKER
NO
DONE/REPEAT
818

Figure 8

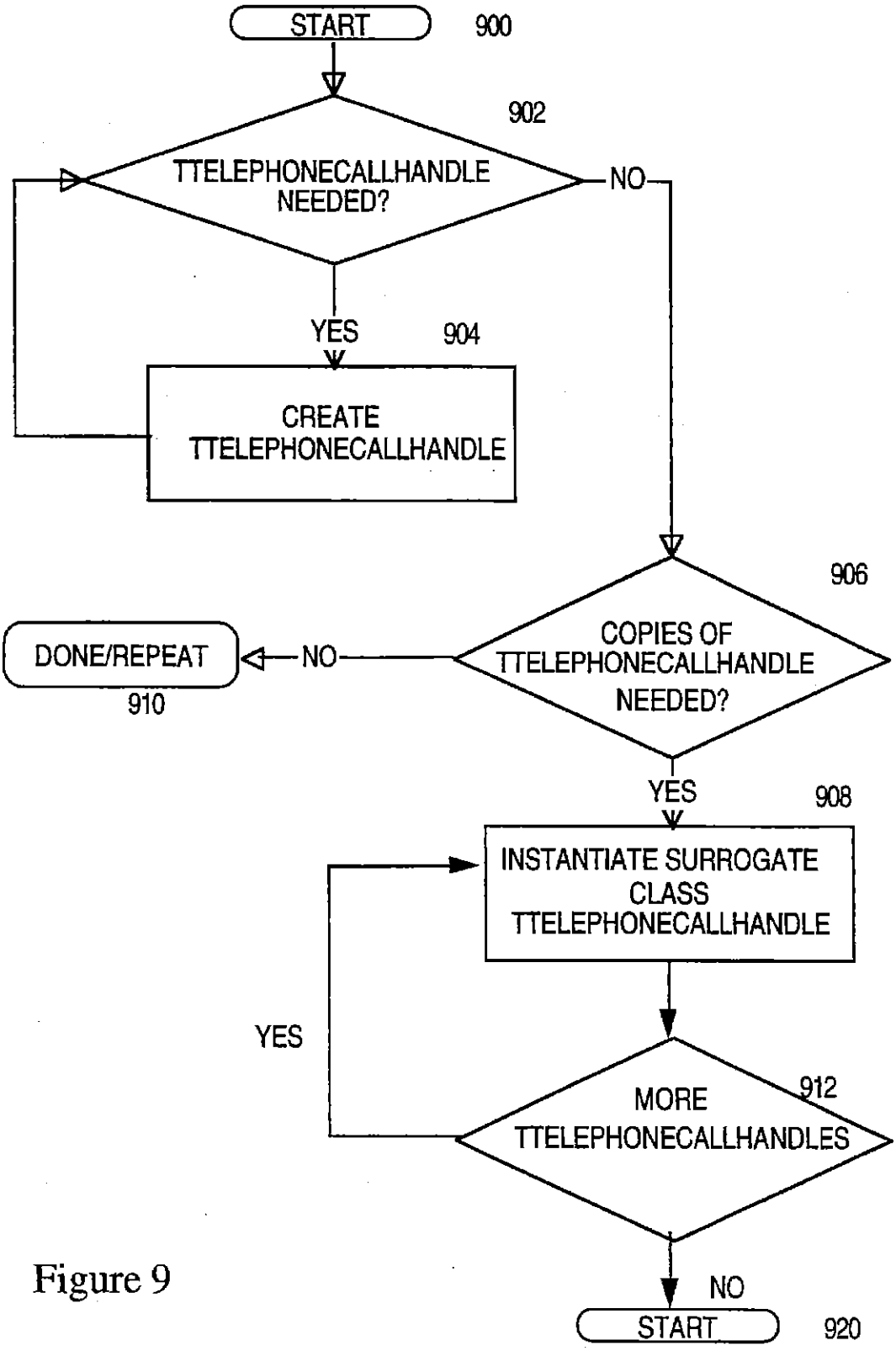

START
900
902
TTELEPHONECALLHANDLE NEEDED?
NO
YES
904
CREATE TTELEPHONECALLHANDLE
906
COPIES OF TTELEPHONECALLHANDLE NEEDED?
DONE/REPEAT
NO
910
YES
908
INSTANTIATE SURROGATE CLASS TTELEPHONECALLHANDLE
YES
MORE TTELEPHONECALLHANDLES
912
NO
START
920

Figure 9

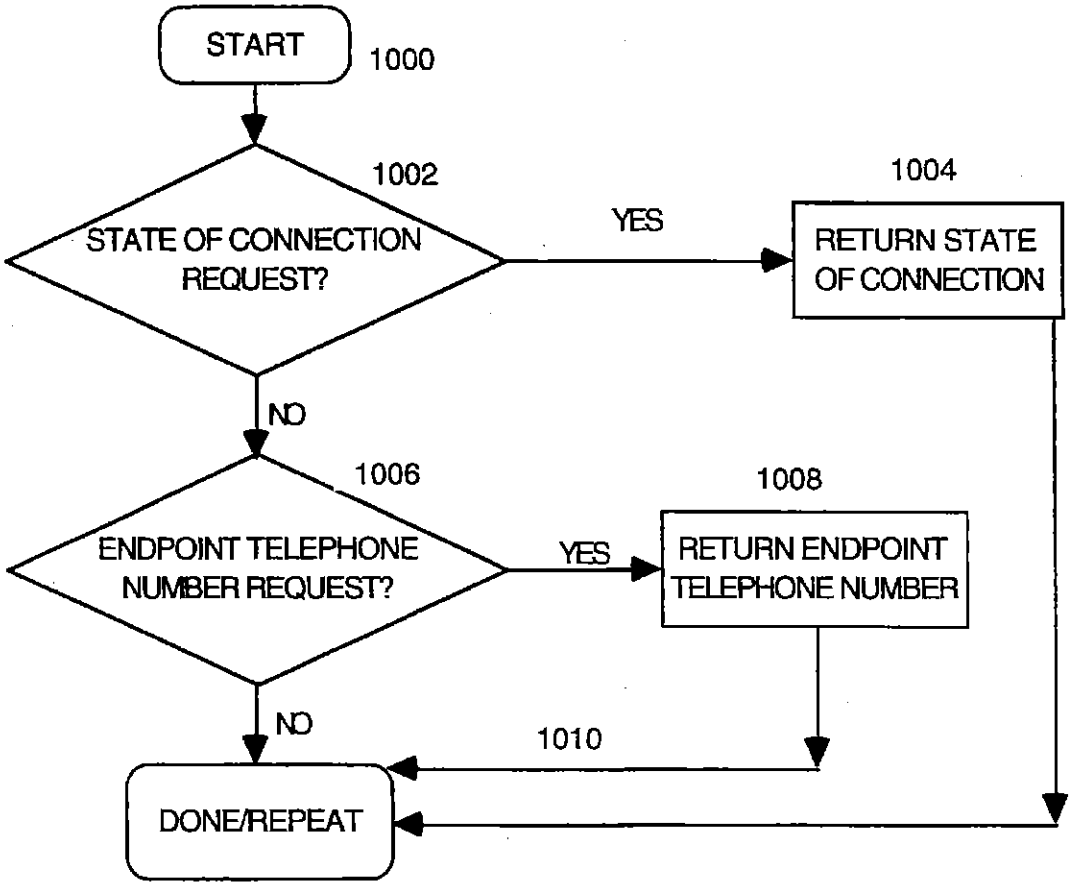

START
1000
1002
STATE OF CONNECTION REQUEST?
YES
1004
RETURN STATE OF CONNECTION
NO
1006
ENDPOINT TELEPHONE NUMBER REQUEST?
YES
1008
RETURN ENDPOINT TELEPHONE NUMBER
NO
1010
DONE/REPEAT

Figure 10

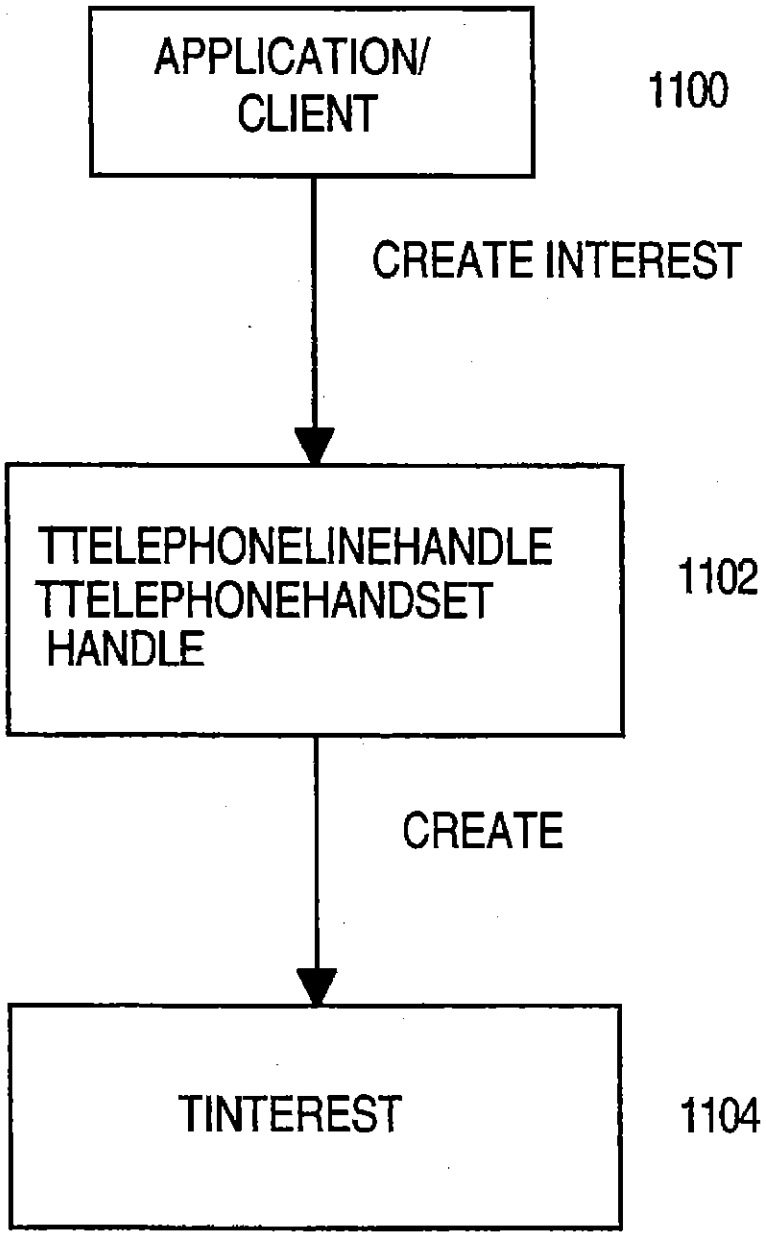

APPLICATION/
CLIENT
1100
CREATE INTEREST
TTELEPHONELINEHANDLE
TTELEPHONEHANDSET
HANDLE
1102
CREATE
TINTEREST
1104

Figure 11

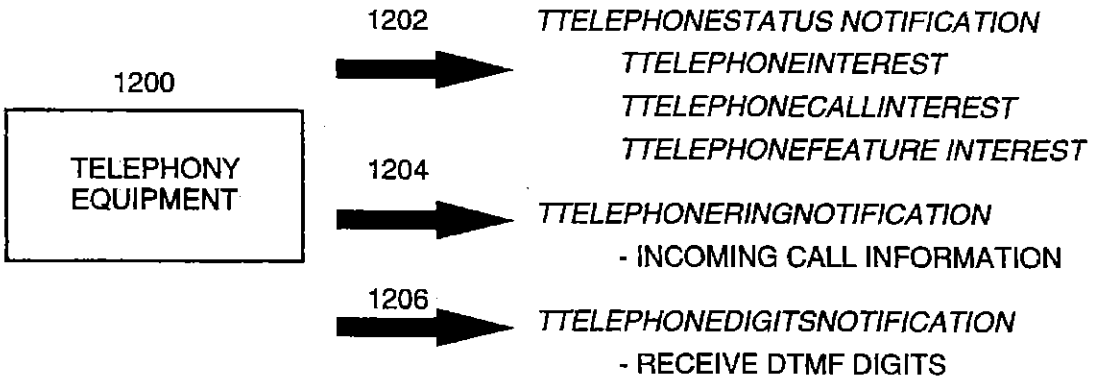

1200
TELEPHONY EQUIPMENT
1202
TTELEPHONESTATUS NOTIFICATION
TTELEPHONEINTEREST
TTELEPHONECALLINTEREST
TTELEPHONEFEATURE INTEREST
1204
TTELEPHONERINGNOTIFICATION
- INCOMING CALL INFORMATION
1206
TTELEPHONEDIGITSNOTIFICATION
- RECEIVE DTMF DIGITS

Figure 12

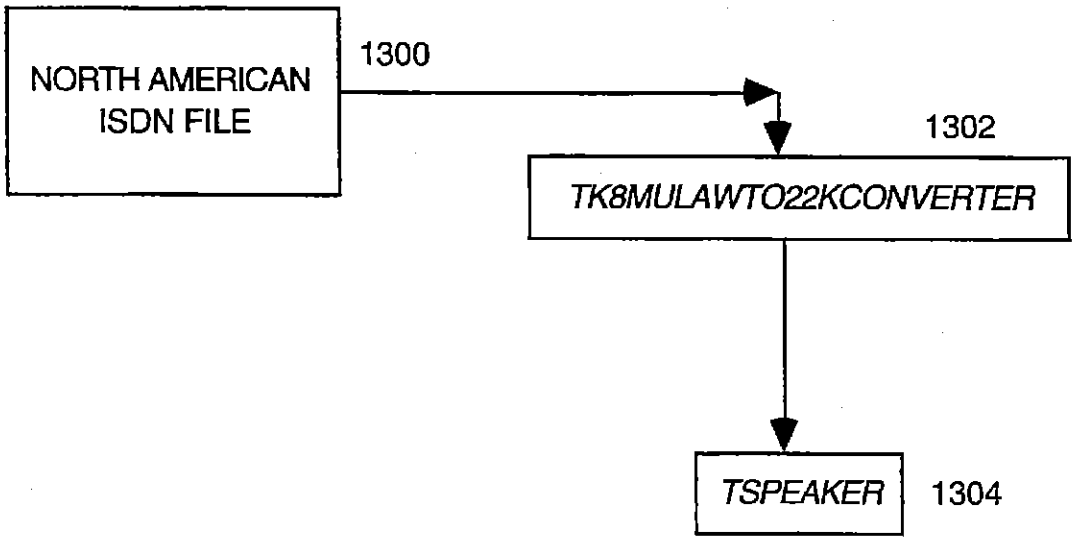

NORTH AMERICAN ISDN FILE
1300
1302
TK8MULAWTO22KCONVERTER
TSPEAKER
1304

Figure 13

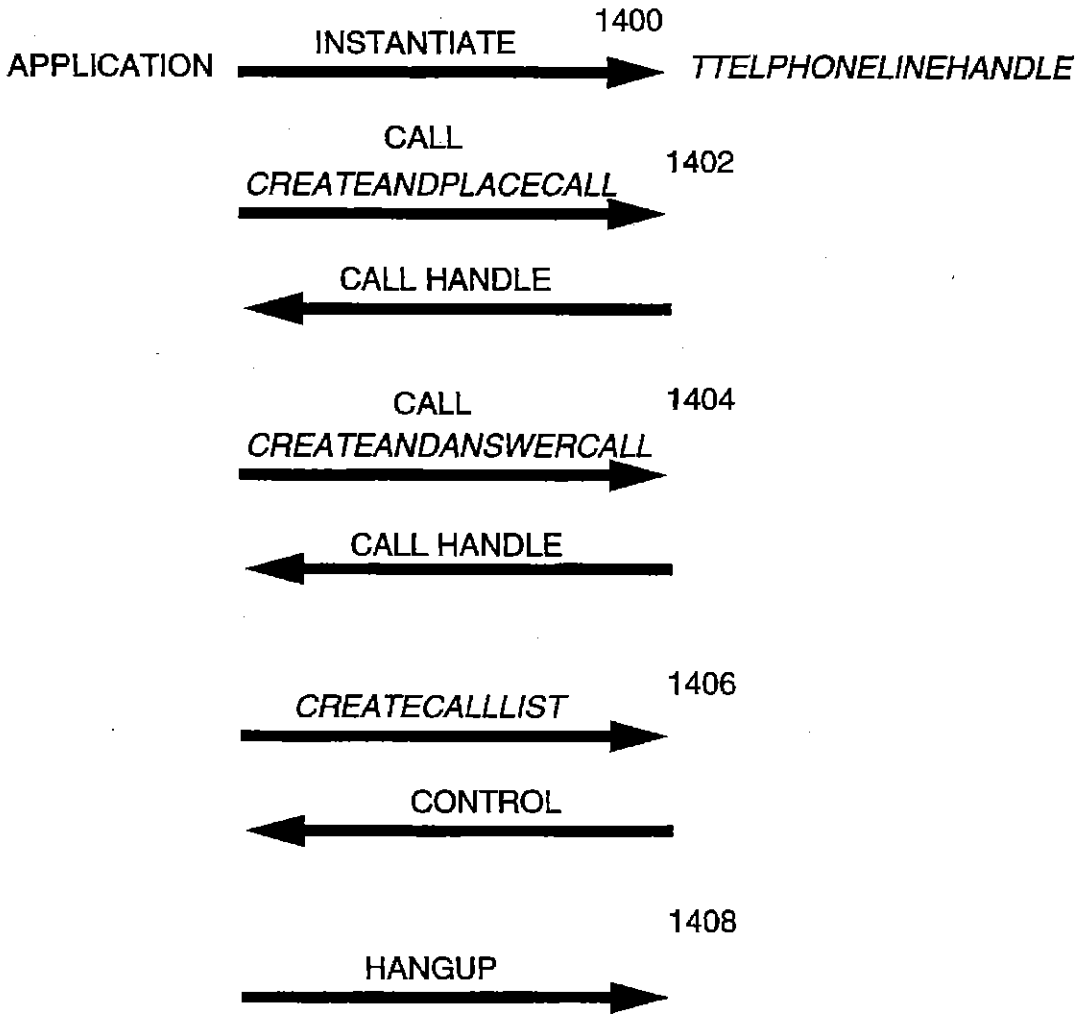

APPLICATION
INSTANTIATE
1400
TTELPHONELINEHANDLE
CALL
CREATEANDPLACECALL
1402
CALL HANDLE
CALL
CREATEANDANSWERCALL
1404
CALL HANDLE
CREATECALLLIST
1406
CONTROL
HANGUP
1408

Figure 14

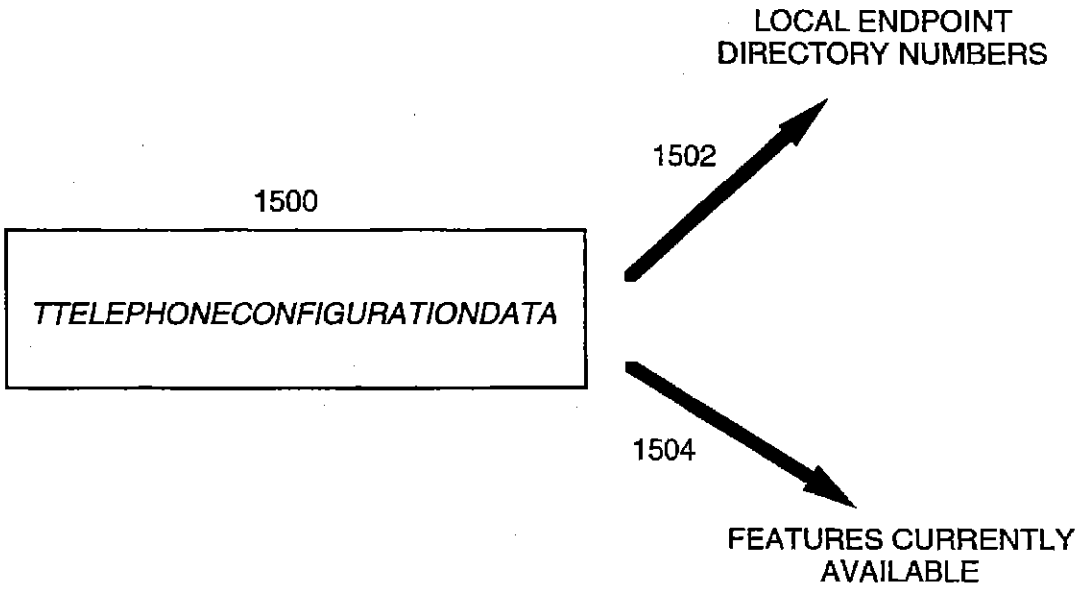

LOCAL ENDPOINT
DIRECTORY NUMBERS
1502
1500
TTELEPHONECONFIGURATIONDATA
1504
FEATURES CURRENTLY
AVAILABLE

Figure 15

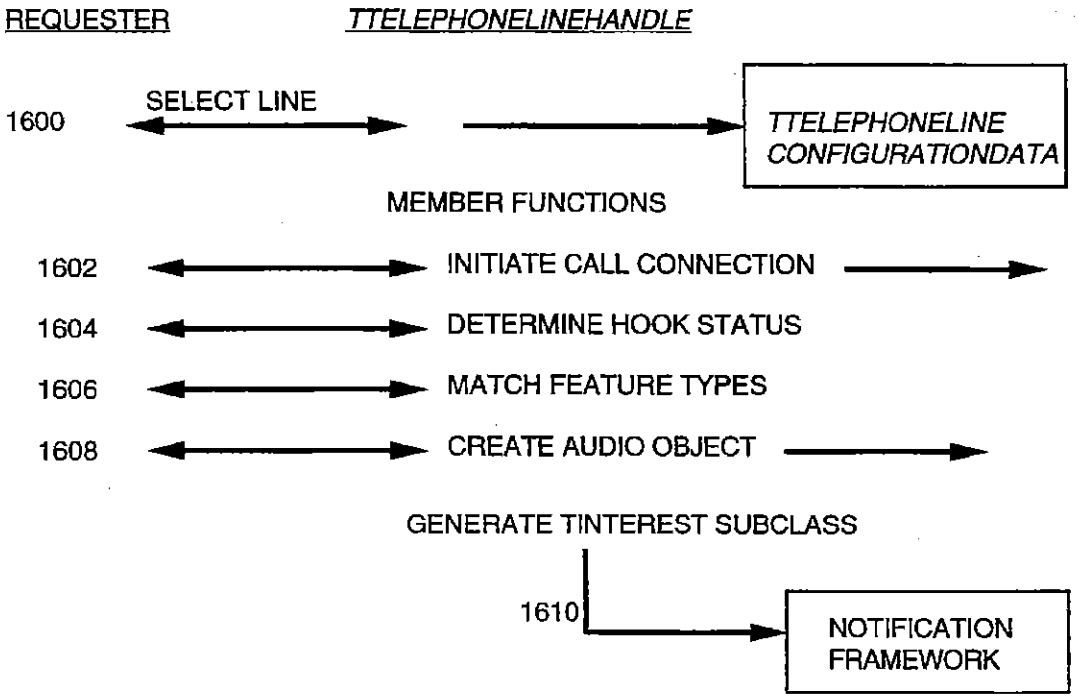

REQUESTER
TTELEPHONELINEHANDLE
1600
SELECT LINE
TTELEPHONELINE CONFIGURATIONDATA
MEMBER FUNCTIONS
1602
INITIATE CALL CONNECTION
1604
DETERMINE HOOK STATUS
1606
MATCH FEATURE TYPES
1608
CREATE AUDIO OBJECT
GENERATE TINTEREST SUBCLASS
1610
NOTIFICATION FRAMEWORK

Figure 16

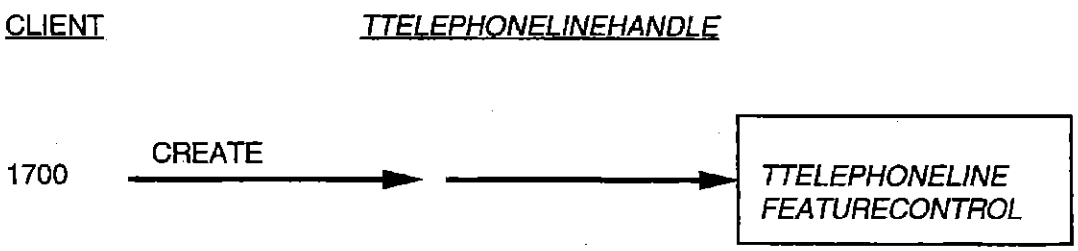


Figure 17

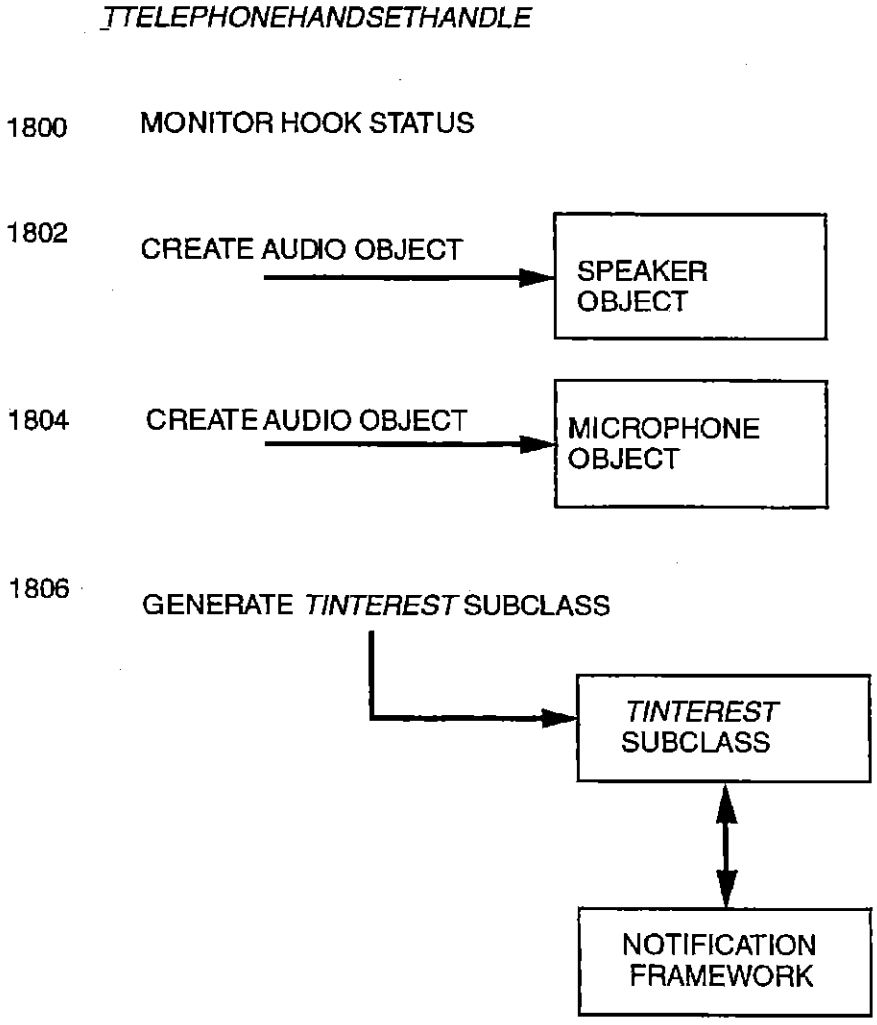


Figure 18

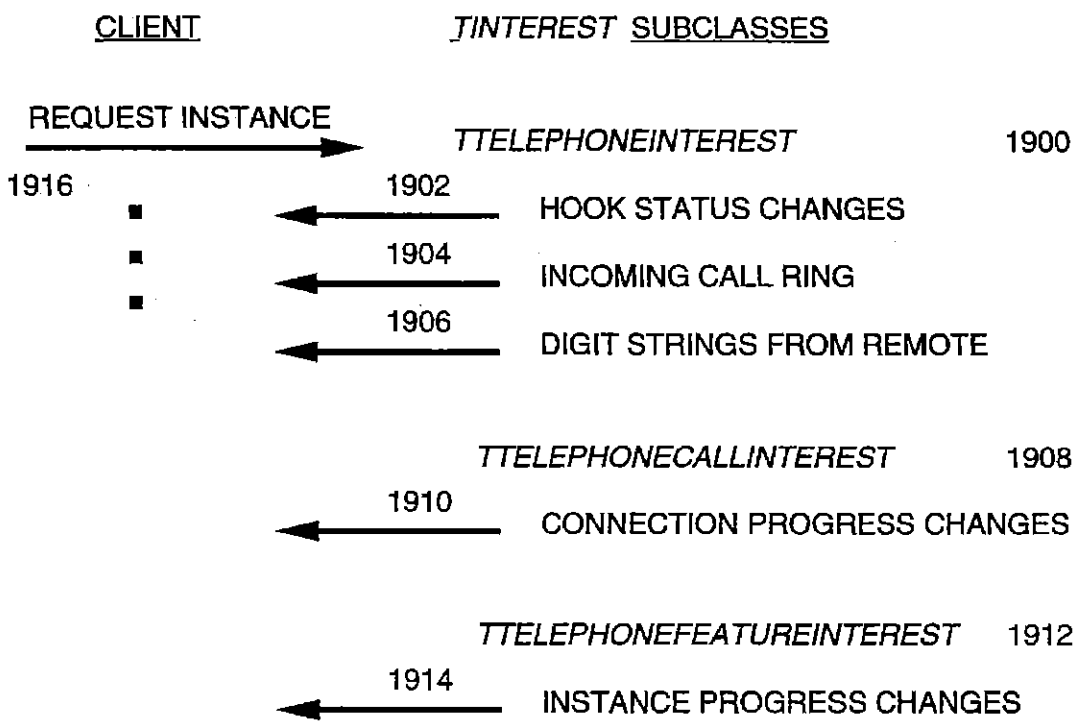

CLIENT
TINTEREST SUBCLASSES
REQUEST INSTANCE
TTELEPHONEINTEREST
1900
1916
1902
HOOK STATUS CHANGES
1904
INCOMING CALL RING
1906
DIGIT STRINGS FROM REMOTE
TTELEPHONECALLINTEREST
1908
1910
CONNECTION PROGRESS CHANGES
TTELEPHONEFEATUREINTEREST
1912
1914
INSTANCE PROGRESS CHANGES

Figure 19

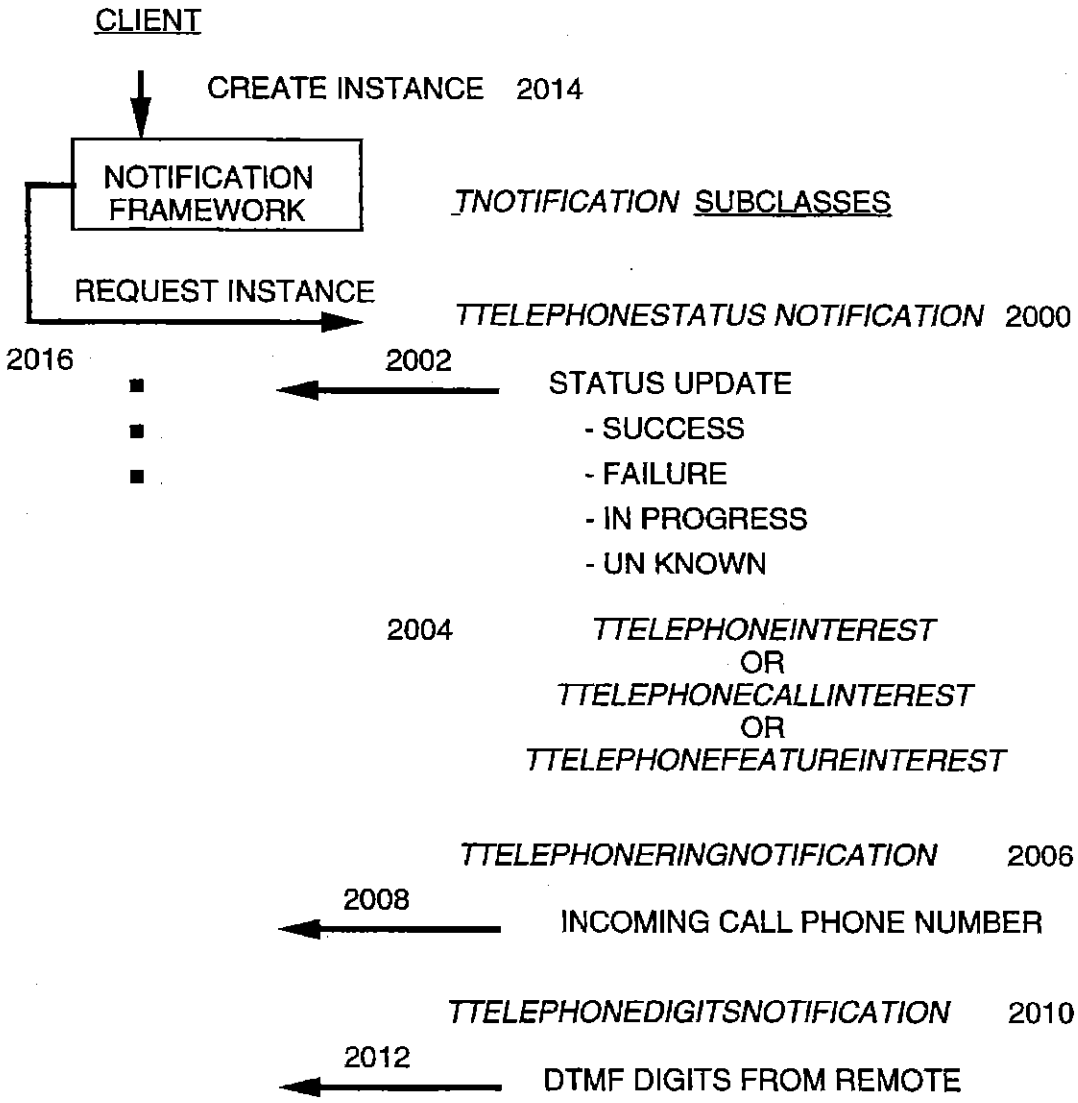

CLIENT
CREATE INSTANCE 2014
NOTIFICATION FRAMEWORK
JNOTIFICATION SUBCLASSES
REQUEST INSTANCE
TTELEPHONESTATUS NOTIFICATION 2000
2016
2002
STATUS UPDATE
- SUCCESS
- FAILURE
- IN PROGRESS
- UN KNOWN
2004
TTELEPHONEINTEREST
OR
TTELEPHONECALLINTEREST
OR
TTELEPHONEFEATUREINTEREST
TTELEPHONERINGNOTIFICATION 2006
2008
INCOMING CALL PHONE NUMBER
TTELEPHONEDIGITSNOTIFICATION 2010
2012
DTMF DIGITS FROM REMOTE

Figure 20

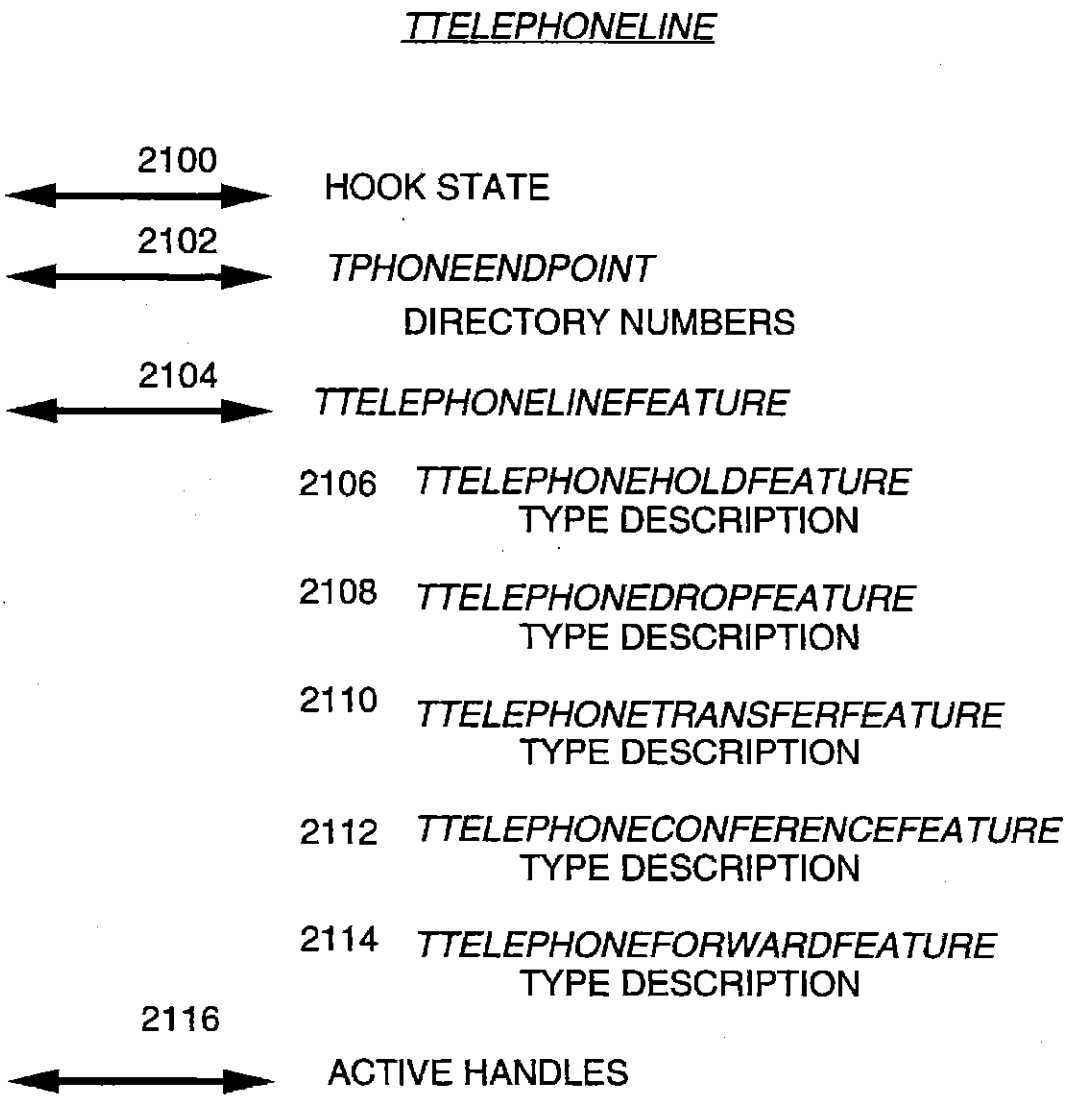

TTELEPHONELINE
2100
HOOK STATE
2102
TPHONEENDPOINT
DIRECTORY NUMBERS
2104
TTELEPHONELINEFEATURE
2106 TTELEPHONEHOLDFEATURE
TYPE DESCRIPTION
2108 TTELEPHONEDROPFEATURE
TYPE DESCRIPTION
2110 TTELEPHONETRANSFERFEATURE
TYPE DESCRIPTION
2112 TTELEPHONECONFERENCEFEATURE
TYPE DESCRIPTION
2114 TTELEPHONEFORWARDFEATURE
TYPE DESCRIPTION
2116
ACTIVE HANDLES

Figure 21

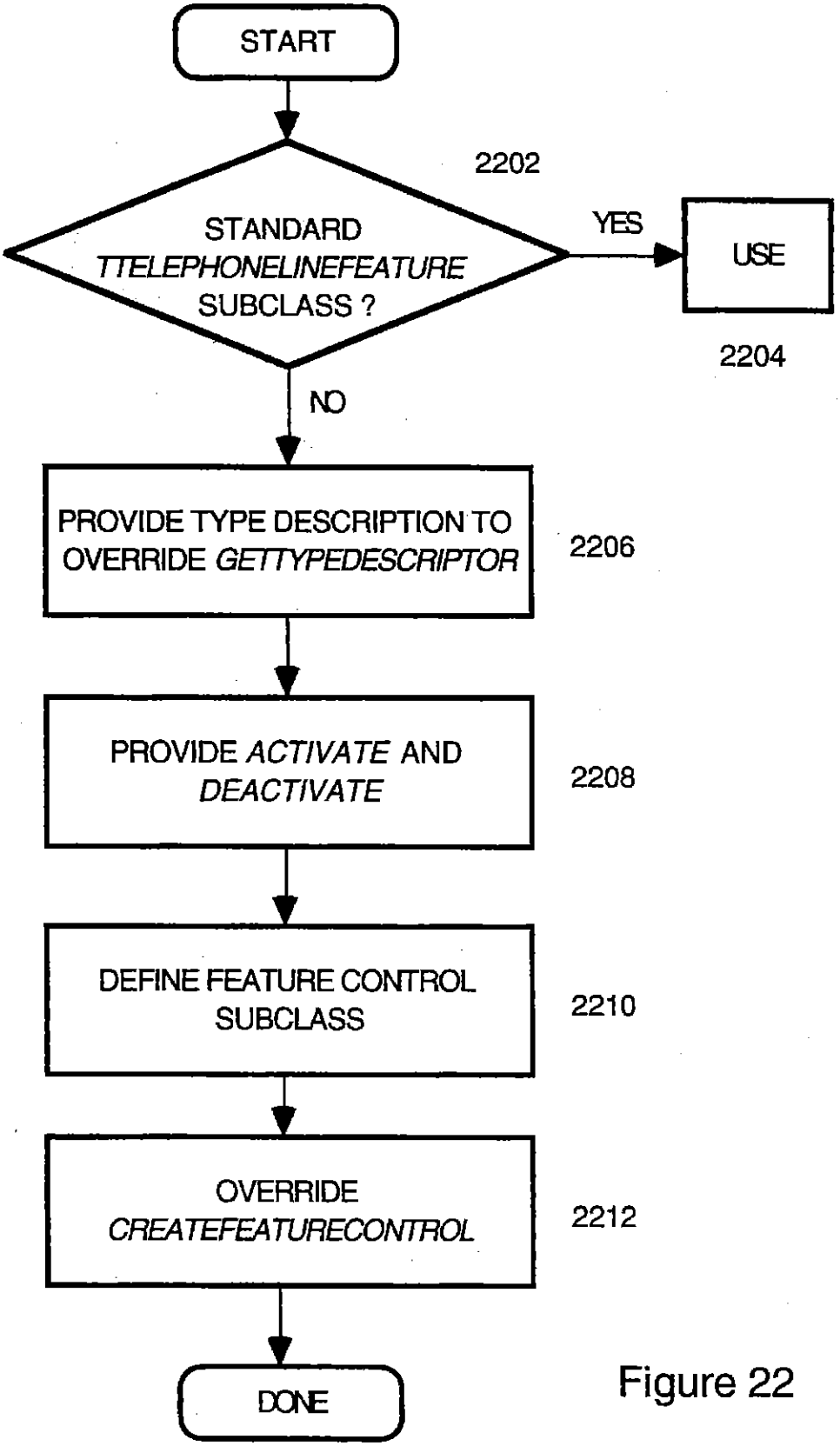

START
2202
STANDARD TTELEPHONELINEFEATURE SUBCLASS ?
YES
USE
2204
NO
PROVIDE TYPE DESCRIPTION TO OVERRIDE GETTYPEDESCRIPTOR
2206
PROVIDE ACTIVATE AND DEACTIVATE
2208
DEFINE FEATURE CONTROL SUBCLASS
2210
OVERRIDE CREATEFEATURECONTROL
2212
DONE

Figure 22

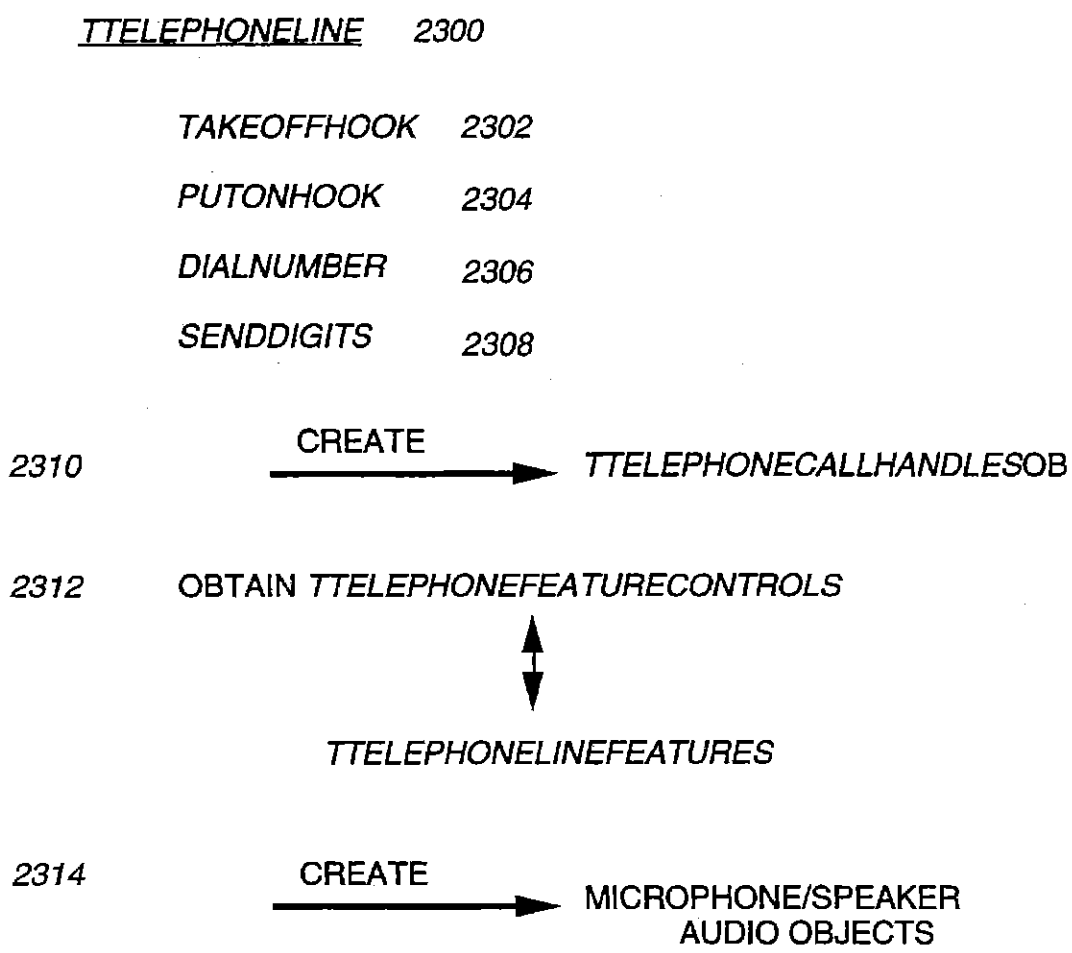

TTELEPHONELINE 2300
TAKEOFFHOOK 2302
PUTONHOOK 2304
DIALNUMBER 2306
SENDDIGITS 2308
2310
CREATE
TTELEPHONECALLHANDLESOB
2312
OBTAIN TTELEPHONEFEATURECONTROLS
TTELEPHONELINEFEATURES
2314
CREATE
MICROPHONE/SPEAKER
AUDIO OBJECTS

Figure 23

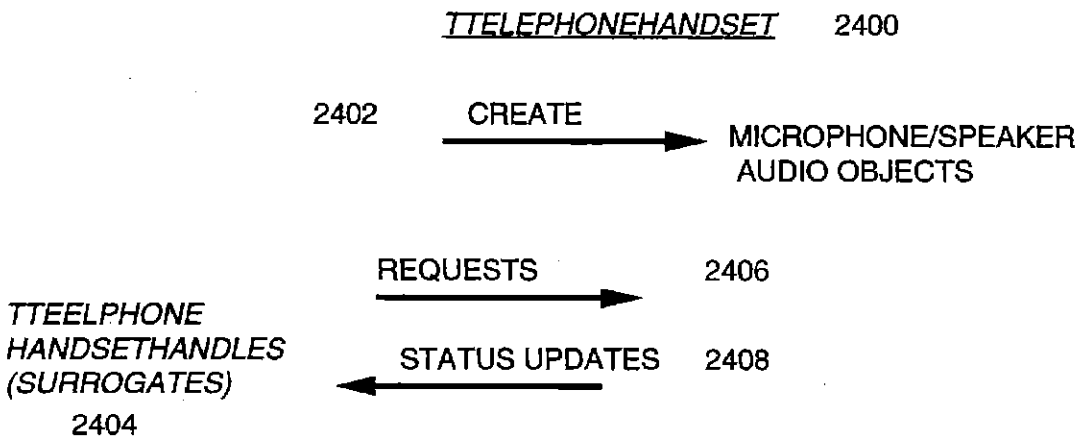


Figure 24

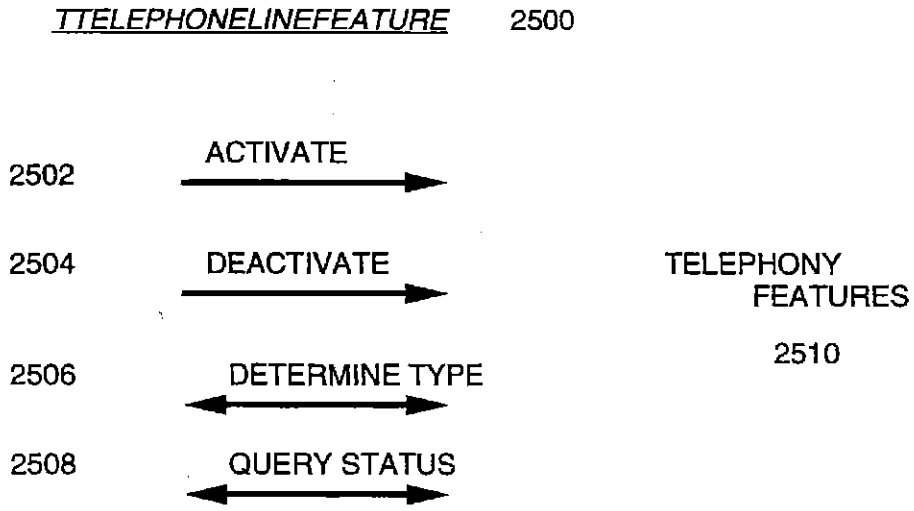


Figure 25

OBJECT-ORIENTED TELEPHONY SYSTEM

COPYRIGHT NOTIFICATION

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to telephony systems and more particularly to a system for interfacing application elements with telephony elements.

2. Description of the Prior Art

Current telephony systems are fairly rudimentary, having changed very little from the basic principles of early telephony. Early systems often utilized switchboard operators for making connections between calling and called parties. By using computer technology, the need for such operators has been reduced to providing assistance in only special circumstances.

Other technological advances have turned telephony systems into an information transfer network or data highway. Facsimile is one of many examples which use the telephony system as an information transfer network. Perhaps the heaviest use of telephony systems comes from simple voice phone calls between calling and called parties. This interaction however, takes least advantage of telephony system capabilities.

Moreover, a typical user only has the most basic of equipment, which is designed for mere real-time voice transmission. The telephony system, however, is capable of transfering other infomation in addition to voice, and is also capable of providing a variety of information transfers which more fully utilize the available bandwidth and network capabilities of the telephony system.

The typical PBX is a classic example of a technology whose power has outstripped the available means of delivery. The average user makes use of only a tiny fraction of the full capability of most modern phone systems. The standard telephone handset is a human-interface bottleneck. At best, it is characterized by rows of buttons which must be labeled because there is no conventional mapping between position and function. Unfortunately, the labels give no clue as to the sequence in which the buttons must be pressed to accomplish a given operation. The presence of multi-function buttons (Transfer/3-Way Conference) adds further confusion to the picture.

While telephone companies have used computer technology to take advantage of, and to advance the potential of, the telephony system, the individual user has not been able to exploit the potential to the same extent. The closest an individual user comes to exploiting the potential of a telephony system is when a phone answering machine is used, and perhaps accessed remotely.

The telephone is really just a primitive terminal. A connection to another terminal is established by lifting the receiver to signal that network services are desired and then inputting a sequence of numbers which identify the terminal connection. The connection is terminated by hanging up. These simple operations provide convenience in creating a simple connection, but other aspects of the phone system require users to keep track of a variety of details.

For example, phone numbers are often memorized or kept manually in a handwritten notebook which must be carried around at all times. Address books, index cards, and massive lists printed on cheap paper and bound in unwieldy tomes are also frequently used. But storage is only part of the problem. Before they can be used, the numbers must be accessed—retrieved from memory, looked up in the card file or directory. And since numbers change over time, they must also be periodically updated. Storing numbers electronically in personal or on-line databases can mitigate the problems of retrieval and maintenance to some extent, but after it is located the number must still be manually transferred to the telephone set.

A telephone set also provides a less than ideal user interface in the case of operations which are more complex than establishing a simple one-to-one connection. In the case of a conference call, for example, the average user needs to check the reference manual to determine what sequence of keys to use: is it "*7" or "*8"? One alternative is to provide a special telephone set equipped with a row or two of extra buttons—the telephony equivalent of function keys on a computer keyboard, except that the labels on telephone buttons are typically a little longer: "TRNSF" for call transfer rather than "F6".

Therefore, there is a need to untap the potential of the present telephony system, especially the information transfer and network capabilities of the system.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide improved access to telephony elements.

It is another object of the present invention to apply computer technology to telephony systems.

It is yet another object of the present invention to provide an interface between telephony system elements and application elements.

It is also an object of the present invention to make the boundary between the telephony world and generic sound world appear smooth and seamless.

The above objects are realized by providing elements which flexibly interface with various telephony system elements. These interface elements typically do not replace the telephone system elements, but rather provide a transparent way of interfacing with particular telephone system elements. The interfacing elements are designed to interface with particular elements of the telephony system, whether the telephony system elements of the telephony system are hardware elements, or merely protocols. For example, an interface element could provide a simple interface for receiving a call, and notifying an application element that a call is pending.

The interface elements provide a flexible and convenient way in which a programmer developing an application program involving telephony can utilize telephony system features without concerning themselves with the details of how the feature is accessed for a particular telephony system. The interface elements are also useful to hardware designers of telephony equipment.

Computer-based telephony has the potential to eliminate the limitations of the phone set while at the same time providing tight integration of telephone-based communications with applications running on the desktop. It is an

essential part of the system foundation required to enable remote, real-time collaboration.

The present invention allows applications to access whatever internal telephone hardware is available on a target machine and provides generic dialing functionality via a modem or other external hardware device.

An object-oriented operating system is an essential element of computer-based telephony applications in a preferred embodiment.

Other objects and advantages of the present invention will become apparent from the following detailed description when viewed with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows a representative system in accordance with a preferred embodiment;

FIG. **2** shows a typical hookup of a handset to a wall jack in accordance with a preferred embodiment;

FIG. **3** shows a handset hooked up to multiple lines and multiple jacks in accordance with a preferred embodiment;

FIG. **4** shows a computer-based telephony system utilizing a modem in accordance with a preferred embodiment;

FIG. **5** shows a computer-based telephony system having a computer interposed between the handset and jack in accordance with a preferred embodiment;

FIG. **6** shows hardware in accordance with a preferred embodiment;

FIG. **7** shows the instantiation of a TTelephoneLineHandle object in accordance with a preferred embodiment;

FIG. **8** shows the functional characteristics of TTelephoneLineHandle. in accordance with a preferred embodiment;

FIG. **9** shows the instantiation of TTelephoneCallHandle in accordance with a preferred embodiment;

FIG. **10** shows the functional characteristics of TTelephoneCatlHandle in accordance with a preferred embodiment;

FIG. **11** shows an application creating an interest in accordance with a preferred embodiment;

FIG. **12** shows a telephony system sending information in accordance with a preferred embodiment;

FIG. **13** shows the playback of a sound file in accordance with a preferred embodiment;

FIG. **14** shows an application interacting with TTelephoneLineHandle to perform various telephony transactions in accordance with a preferred embodiment;

FIG. **15** shows configuration data in accordance with a preferred embodiment;

FIG. **16** shows the functions of TTelephoneLineHandle in accordance with a preferred embodiment;

FIG. **17** shows the functions of TTelephoneLineFeature-Control in accordance with a preferred embodiment;

FIG. **18** shows the functions of TTelephoneHandsetHandle in accordance with a preferred embodiment;

FIG. **19** shows TInterest subclasses in accordance with a preferred embodiment;

FIG. **20** shows TNotification subclasses in accordance with a preferred embodiment;

FIG. **21** shows the functional characteristics of TTelephoneLine in accordance with a preferred embodiment;

FIG. **22** shows a development cycle for a designer of telephony equipment in accordance with a preferred embodiment;

FIG. **23** shows the functional characteristics of TTelephoneLine in accordance with a preferred embodiment;

FIG. **24** shows the functional characteristics of TTelephoneLineHandset in accordance with a preferred embodiment;

FIG. **25** shows the functional aspects of TTelephoneLineFeature in accordance with a preferred embodiment;

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The detailed embodiments of the present invention are disclosed herein. it should be understood, however, that the disclosed embodiments are merely exemplary of the invention, which may be embodied in various forms. Therefore, the details disclosed herein are not to be interpreted as limiting, but merely as the basis for the claims and as a basis for teaching one skilled in the art how to make and/or use the invention.

As used herein, "telephony system" includes all aspects of telephonic communication.

Telephony Standards Networks

Historically, telephone networks standards have been established on a country-by-country basis. In many cases they have been subject to considerable variation, creating a localization nightmare, especially in Europe. Because of emerging digital standards such as ISDN (Integrated Services Digital Network) and the formation of the European Community in 1992, this situation will likely be ameliorated in the near future. In the United States, where incentives to move rapidly to ISDN are not as strong, we will very likely have to contend with two parallel networks, one analog and one digital, for some time to come.

In the analog domain, computer control of telephony functions is typically performed via a modem. (Note that the modem need not necessarily be a piece of external hardware. RISC and chip-based Digital Signal Processing technology make possible the implementation of fully functional internal software modems.) A widespread de facto standard for computer-to-modem communication is the so-called "AT" command set originally defined by Hayes, Inc.

In the digital domain the ISDN standard has been developed by a multinational working group of the CCITT (International Telegraph and Telephone Consultative Committee). The ISDN network layer protocols for setting up and tearing down connections are defined in CCITT Recommendation Q.931. (ISDN also includes specifications for physical and data link layers, as well as for a higher-level management entity.)

Switches

Between the telephone network and the telephone handset lies a switch of some sort which links a specific extension to the network. In the commercial market, this switch is most likely a PBX (Private Branch exchange). Aside from providing simple connections via the network, switches typically provide a variety of so-called supplementary services, such as holding, transferring, and conferencing. The protocols for accessing these services not only vary significantly, but have historically been proprietary to each switch manufacturer. While the ISDN network layer specification in Q.931 does address supplementary features, considerable

latitude remains in the choice of features and in the actual sequence of messages used to implement a given feature.

Audio Data Formats

Depending on the type of network and on the interface between computer and telephone network, it may be possible to record audio data off the phone line for storage and playback on the computer. In the case of analog data, a pathway to Analog to Digital Conversion services would have to be provided, and the data could then be represented using whatever internal standards are valid for a given design center. In the case of digital data, the matter of external standards arises. CCITT has defined several data format standards compatible with the ISDN B-Channel transfer rate of 64,000 bits per second. There are 2 standards for uncompressed voice signals which take advantage of the full available bandwidth. Both of these are based on log-companded PCM (Pulse Code Modulation). In the first case, which is the primary format for North America, log-companding is provided by the It-Law function, in the second, which has been adopted primarily in Europe, the A-Law function is used. Each assumes an 8 KHz sample rate and allocates 8 bits per sample, providing an effective signal bandwidth of 3.5 KHz.

A number of standards for audio compression have been adopted by CCITT on the one hand and by the digital cellular industry on the other.

Object-Oriented Programming

In a preferred embodiment, the invention is implemented in the C++ programming language using object-oriented programming techniques. As will be understood by those skilled in the art, Object-Oriented Programming (OOP) objects are software entities comprising data structures and operations on the data. Together, these elements enable objects to model virtually any real-world entity in terms of its characteristics, represented by its data elements, and its behavior, represented by its data manipulation functions. In this way, objects can model concrete things like people and computers, and they can model abstract concepts like numbers or geometrical concepts. The benefits of object technology arise out of three basic principles: encapsulation, polymorphism and inheritance.

Objects hide, or encapsulate, the internal structure of their data and the algorithms by which their functions work. Instead of exposing these implementation details, objects present interfaces that represent their abstractions cleanly with no extraneous information. Polymorphism takes encapsulation a step further. The idea is many shapes, one interface. A software component can make a request of another component without knowing exactly what that component is. The component that receives the request interprets it and figures out according to its variables and data, how to execute the request. The third principle is inheritance, which allows developers to reuse pre-existing design and code. This capability allows developers to avoid creating software from scratch. Rather, through inheritance, developers derive subclasses that inherit behaviors, which the developer then customizes to meet their particular needs.

A prior art approach is to layer objects and class libraries in a procedural environment. Many application frameworks on the market take this design approach. In this design, there are one or more object layers on top of a monolithic operating system. While this approach utilizes all the principles of encapsulation, polymorphism, and inheritance in the object layer, and is a substantial improvement over procedural programming techniques, there are limitations to this approach. These difficulties arise from the fact that while it is easy for a developer to reuse their own objects, it is difficult to use objects from other systems and the developer still needs to reach into the lower non-object layers with procedural Operating System (OS) calls.

Another aspect of object-oriented programming is a framework approach to application development. One of the most rational definitions of frameworks come from Ralph E. Johnson of the University of Illinois and Vincent F. Russo of Purdue. In their 1991 paper, *Reusing Object-Oriented Designs*, University of Illinois tech report UIUCDCS91-1696 they offer the following definition: "An abstract class is a design of a set of objects that collaborate to carry out a set of responsibilities. Thus, a framework is a set of object classes that collaborate to execute defined sets of computing responsibilities." From a programming standpoint, frameworks are essentially groups of interconnected object classes that provide a pre-fabricated structure of a working application. For example, a user interface framework might provide the support and "default" behavior of drawing windows, scrollbars, menus, etc. Since frameworks are based on object technology, this behavior can be inherited and overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise. This is a major advantage over traditional programming since the programmer is not changing the original code, but rather extending the software. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling but at the same time frees them to then supply the specific actions unique to the problem domain.

From a business perspective, frameworks can be viewed as a way to encapsulate or embody expertise in a particular knowledge area. Corporate development organizations, Independent Software Vendors (ISV)s and systems integrators have acquired expertise in particular areas, such as manufacturing, accounting, or currency transactions as in our example earlier. This expertise is embodied in their code. Frameworks allow organizations to capture and package the common characteristics of that expertise by embodying it in the organization's code. First, this allows developers to create or extend an application that utilizes the expertise, thus the problem gets solved once and the business rules and design are enforced and used consistently. Also, frameworks and the embodied expertise behind the frameworks have a strategic asset implication for those organizations who have acquired expertise in vertical markets such as manufacturing, accounting, or bio-technology would have a distribution mechanism for packaging, reselling, and deploying their expertise, and furthering the progress and dissemination of technology.

Historically, frameworks have only recently emerged as a mainstream concept on personal computing platforms. This migration has been assisted by the availability of object-oriented languages, such as C++. Traditionally, C++ was found mostly on UNIX systems and researcher's workstations, rather than on Personal Computers in commercial settings. It is languages such as C++ and other object-oriented languages, such as Smalltalk and others, that enabled a number of university and research projects to produce the precursors to today's commercial frameworks and class libraries. Some examples of these are InterViews from Stanford University, the Andrew toolkit from Carnegie-Mellon University and University of Zurich's ET++ framework.

There are many kinds of frameworks depending on what level of the system you are concerned with and what kind of problem you are trying to solve. The types of frameworks range from application frameworks that assist in developing the user interface, to lower level frameworks that provide basic system software services such as communications, printing, file systems support, graphics, etc. Commercial examples of application frameworks are MacApp (Apple), Bedrock (Symantec), OWL (Borland), NeXTStep App Kit (NEXT), and Smalltalk-80 MVC (ParcPlace) to name a few.

Programming with frameworks requires a new way of thinking for developers accustomed to other kinds of systems. In fact, it is not like "programming" at all in the traditional sense. In old-style operating systems such as DOS or UNIX, the developer's own program provides all of the structure. The operating system provides services through system calls—the developer's program makes the calls when it needs the service and control returns when the service has been provided. The program structure is based on the flow-of-control, which is embodied in the code the developer writes.

When frameworks are used, this is reversed. The developer is no longer responsible for the flow-of-control. The developer must forego the tendency to understand programming tasks in term of flow of execution. Rather, the thinking must be in terms of the responsibilities of the objects, which must rely on the framework to determine when the tasks should execute. Routines written by the developer are activated by code the developer did not write and that the developer never even sees. This flip-flop in control flow can be a significant psychological barrier for developers experienced only in procedural programming. Once this is understood, however, framework programming requires much less work than other types of programming.

The invention is preferably practiced in the context of an operating system resident on a personal computer such as the IBM® PS/2 ® or Apple® Macintosh® computer. A representative hardware environment is depicted in FIG. 1, which illustrates a typical hardware configuration of a computer in accordance with the subject invention having a central processing unit **100**, such as a conventional microprocessor, and a number of other units interconnected via a system bus **132**. The computer shown in FIG.1 includes a Read Only Memory (ROM) **104**, a Random Access Memory (RAM) **106**, an I/O adapter **112** for connecting peripheral devices such as disk units **108** and other I/O peripherals represented by **110** to the system bus **132**, a user interface adapter **128** for connecting a keyboard **130**, a mouse **126**, a speaker **122**, a microphone **124**, and/or other user interface devices such as a touch screen device (not shown) to the bus, a communication adapter **116** for connecting the workstation to a data processing network represented by **114**. A display adapter **120** for connecting the bus to a display device **118**. The workstation has resident thereon an operating system such as the Apple System/7 ® operating system.

Computer-Based Telephony Applications

Voice-oriented applications of computer-based telephony fall into 3 categories depending on their basic requirements:

1) Applications which rely on software control of telephony features without provisions for capturing or accessing the voice signal;

2) Applications which do need to record and play back voice data; and

3) Applications which assume simultaneous transmission of voice and non-voice data.

Over the years, a variety of devices have been devised to make the telephone more useful. Many of these involve computer-controlled telephony.

Virtual Telephone Set

Applications of this genre directly address the issue of the advanced telephone system which is so complex that it is used to its full potential by less than 10% of its users. The basic problem with such systems has to do with the fact that the average telephone handset is a device with a very low interface bandwidth. Each new feature added to the phone systems results in another button on the handset (or—shades of MS-DOS—another special sequence of numbers to memorize: "*3" to transfer, "*5" to forward).

Software control of such functions from the desktop, combined with a well-designed graphical user interface, will help end-users to bypass the arcane idiosyncrasies of their local PBX and enable them to start benefiting from the technological improvements that telephone equipment manufacturers have managed to achieve since the days of Alexander Graham Bell. A good example of this concept is Northern Telecom's Meridian TeleCenter, which runs on current Macintoshes, but of course operates only with Northern Telecom's own PBX's.

Telemarketing

In it's simplest form, computer-controlled telemarketing tools simply permit automatic dialing from a database of telephone numbers. More sophisticated variants dial out on multiple lines and detect when a human being has answered on the other end of the connection, or present detailed customer information on the screen for each dialed party.

Voice I/O

Desktop Answering Machine

The idea here is to place Voice Mail on the desktop in order to bring it up to the level of functionality we have come to expect from text mail systems. The typical Voice Mail system today is a centralized service which is parallel to and completely isolated from all other forms of electronic communication.

Even when they are available, such features as group addressing and on-line directories are difficult to use via a standard push-button phone set and must be set up and maintained separately from electronic mail databases even though they contain duplicate information. Calling in to check one's voice mailbox is just one more unnecessary detour in the flow of information through office and home. The Voice Mail systems of this decade are analogous to the centralized computer systems of the Seventies.

Some of the improvements to existing telephone services which are just now becoming widely available will be greatly enhanced by the migration of voice messaging to the desktop. For example, the ability for the receiving party to examine the phone number from which an incoming call was placed (so-called Calling Party Identification) will, among other things, permit the user to set up personalized greeting messages. For example, if I am expecting an important call from my brother, John, and I know I may not be in when he calls, I can record a message to be played when my computer answers any call originating from one of the phone numbers listed for him in my electronic address book.

The presence of a computer on the receiving end of a call also increases the potential for manipulating messages once they have been recorded. Any editing of audio data is obviously out of the question if one's only terminal is a push-button handset. With the increasing use of digital formats for transmission of voice data (as with ISDN) it is extremely wasteful not to provide a means for capturing the audio signal as it streams by.

Remote Access

To access your office computer system from home, all you need is a phone line, a modem and another computer. So what if you're at the airport and you realize you forgot to bring the address of the customer site you are going to visit? What if you arrive at your destination to find that the airline has lost your modem along with the rest of your luggage? What if you can't afford to buy a second computer? Then you need the ability to dial up your office system from a standard phone and access the information stored on your hard disk by asking the computer to read it to you. You'll also need Text-to-Speech Synthesis software on your office machine and a way to control the desktop using DTMF (dual-tone multi-frequency) keys or voice commands (i.e. Automatic Speech Recognition).

Interactive Voice Response

This term covers a broad range of applications which permit remote callers to listen to selections from a collection of prerecorded messages. These applications may be solely dedicated to information delivery (e.g. the IRS's Teletax service), or they may combine informational interactions with transaction processing (e.g. an order-entry service for a mail-order catalog or a bank-by-phone service which permits depositors to check account balances, transfer funds, and make loan payments).

Traditionally, putting together even a straightforward information delivery system has required a specialized team of programmers and systems integrators. At least one third party providing such services has developed a graphically-oriented tool on the Macintosh for generating a set of control files which are then uploaded to a stand-alone Voice Response system

Voice Bulletin Board Service

This is a voice-mediated version of the familiar electronic BBS. It is actually a hybrid of Voice Mail and Interactive Voice Response. Callers browse through various categories of voice messages left by other callers, then record their own messages in response.

Simultaneous Voice and Data

Collaboration

The role that telephony services play in collaboration depends, of course, on the kind of collaboration we are talking about. Plain vanilla Voice Mail might be involved in sequential, asynchronous collaboration (e.g. reviewer receives copy of document, reads it privately, and responds with a voice message). Simultaneous, asynchronous collaboration involves placing a call to an author of a document (automatically extracting the phone number from a "business card" object attached to the document) so that a reviewer can deliver comments in person while the author is displaying a separate copy of the document. A third type of collaboration is simultaneous, real-time collaboration a l`a CHER, in which a document is simultaneously updated by author(s) and reviewer(s). The degree to which collaboration of this last type enhances the interaction of team members will no doubt vary in inverse proportion with their physical proximity to one another. It is in the case of remote collaboration among geographically dispersed work-groups that telephony comes into play. The basic idea would to maintain a standard voice connection while simultaneously transmitting and receiving document updates over a data connection. Such an interaction could be carried out over two standard phone lines or over a single ISDN connection.

Cheap Video Phone

Another intriguing application of simultaneous voice and data transmission involves the use of an inexpensive video camera along with a video digitizer card to transform a standard ISDN connection into a poor man's video phone. In other words, a voice connection can be enhanced by the periodic bi-directional transmission of captured video frames showing the callers' faces and/or objects held up to the video camera. Even when the frame-rate is extremely low, the end result can be quite interesting.

The present invention has been designed to empower application writers to blaze new trails in all three areas.

Telephony Objects

The present invention is designed to enable the development of computer-based telephony applications, and allow telephony designers to exploit these newly developed applications. The telephone is in essence a primitive terminal used to establish a connection to another terminal by lifting the receiver to signal that network services are desired. A sequence of numbers identifying the terminal you wish to connect with is then input. The connection is broken by hanging up. The present invention supports computer-based telephony by providing a high-level interface to voice-oriented telephone functions. These functions include the "lowest common denominator" of the capabilities in typical telephony configurations. The invention is intended to insulate application-level code from differences in the equipment through which these functions are accessed. At the same time, the invention provides a flexible framework which can be easily extended to include new features or to accommodate the peculiarities of various combinations of telephone sets, local hardware and software platforms, network types, and switch signaling protocols.

When such a flexible framework for interfacing is provided, numerous application program writers will develop software for exploiting the usefulness of such an interface. In turn, telephony system designers will desire to exploit the new application elements, and will therefore also need interface elements to work with. Therefore, two types of objects will be useful to discuss: objects for application writers on the one hand, and objects for developers of extensions to the basic interface on the other. In general, objects of interest to application developers will also be of interest to telephony system developers. Each category will be discussed below. It should be kept in mind, however, that the headings below are for purposes of discussion only, and are not considered to be limiting in any way.

I. Objects for application developers

A telephony system is comprised of many elements which would be useful to interface with. In general, there are two categories of these elements. The first category involves the physical equipment which is used tin implementing a tele-

phony system, while the second involves the transactions which are processed by the telephony system. Objects for each category will be discussed, and examples will be provided outlining possible types of interfacing elements.

A. Objects for hardware-type elements

From the point of view of the application writer, the present invention facilitates the implementation of simple and intuitive user interfaces for computer-based telephony applications. Hardware-independence of the objects will allow an application to run regardless of the phone system attached to the computer or the telephony features currently available to the end-user. A standard protocol allows an application to determine at launch time which of the features it "knows about" are active and which are inactive.

FIG. **2** shows the real-world telephony object most familiar to the end user: the single-line telephone handset **200** which plugs into a wall jack **204** to connect to a telephone line **202**. The fact that there is an enormous "netscape" of wire, fiber, switches and central offices behind the wall jack does not enter into the consciousness of the average user. What is important is that a specific line **202** is now accessible via a specific handset **200**. The correspondence between handsets and lines, however, need not be one-to-one.

A single phone line may be split into several extensions, each with its own handset (not shown) or, as illustrated by FIG. **3**, a single handset **300** may provide sequential access to several lines **302**, each connected to separate jacks **304**. The former case is typical of residential installations. The multiline handset, on the other hand, might be used by a small business or by an administrative assistant supporting 3 executives. Typically, the end-user selects a line by pushing a labeled button somewhere on the handset **300**.

FIGS. **2** and **3** are not labeled as prior art because handset **200** could be implemented to include the features of the present invention.

FIG. **4** represents a possible system for implementing computer-based telephony in accordance with the present invention. In the actual equipment configuration required for computer-based telephony, the telephone line **404** connects not to the handset **400** but to a computer **412** via the medium of an external device, such as a modem **408**. Line **410** connects the computer **412** to modem **408**, and line **404** connects modem **408** to jack **406**. The handset **400**, which is not even necessary for many applications, also connects to the external device **408** or telephone interface, and can usually be used to initiate or terminate connections independently of the computer **412**. The computer may or may not be able to control and/or query the handset.

The computer **412** may be similar to that shown in FIG. **1**, comprising a personal computer or even a workstation. Computer **412** may include a keyboard **414**, display **416**, and floppy drive **418**.

FIG. **5** demonstrates a variation of the system of FIG. **4**, showing a computer **508** which has a telephone interface for connecting handset **500** to jack **506** via lines **502** and **504**, which can be built-in or card based. Computer **508**, like that shown in FIG. **4**, includes a keyboard **512**, display **512**, and floppy drive **514**. Computer **508** is capable of responding to both handset **500** and signals coming over the telephone lines via line **502**. Computer **508** is also capable of passing signals between handset **400** and jack **506** without modification.

The above discussion outlines some specific hardware which could be used to implement the present invention. It should be kept in mind, however, that such hardware is merely illustrative, and that the present invention could be implemented on a variety of systems and devices connected to the telephone network. FIG. **6** demonstrates a device embodying at least some of the principles of the present invention. Device **600** utilizes objects for interfacing with various elements of the telephony system **604** to which it is connected via line **602**. Device **600** may or may not be connected to one or more devices **606**.

It should also be noted that device **600** may advantageously be connected at points in the system other than to a jack. That is, the object-oriented device may be connected out in the network, interfacing with elements of the telephony system.

The system the present invention is implemented on should be capable of multitasking or multiprocessing. This would provide optimum performance of applications and use of system resources via the objects.

ARCHITECTURAL OVERVIEW

The applications running on the devices connected to the telephone system utilize specially designed interface elements to interact with the elements of the telephony system. It should be noted that the terminology "applications running" is meant to include not only software programs which are running on a computer, but also hardware which may be performing particular applications. These interface elements standardize the methods used for passing information from and to various of the telephony system elements. The interface elements are advantageously implemented using objects in a preferred embodiment.

Telephone Line Handle

FIG. **7** is a flow chart, beginning at step **700**, demonstrating the instantiation of multiple TTelephoneLineHandles. As shown by steps **702** and **704**, the present invention views the telephone line as a system resource which can be shared among voice-oriented applications, as long as it has not been previously reserved for data communications. Such applications actually access telephony services through the surrogate class TTelephoneLineHandle (step **706**). In order to permit the user to maintain operation of an answering machine application, for example, in the background while launching repeated instances of, say, an auto-dial application in the foreground, multiple TTelephoneLineHandles may be instantiated for the same line and distributed across team boundaries (steps **706** and **708**). The process of instantiation ends at step **710** when no further TTelephoneLineHandles are necessary, but may be repeated each time an instance of TTelephoneLineHandle is needed.

FIG. **8** shows a possible flow of operations characteristic of the functions carried out by TTelephoneLineHandle. In addition to permitting the state of the telephone line to be modified (steps **802** and **804**) and queried (steps **806** and **808**), TTelephoneLineHandle provides member functions for creating a standard TMicrophone (steps **810** and **812**) or TSpeaker (steps **814** and **816**) object representing the line's audio input or output capability, if any. The functional flow of TTelephoneLineHandle is delineated by steps **800** and **818**.

Note that from the point of view of the voice application, it is immaterial whether the "real" telephone line is connected to the computer via the physical modem device illustrated in FIG. **4**, or via some type of internal interface,

as in FIG. 4. All the application "sees" under either configuration is a TTelephoneLineHandle.

Telephone Handset Handle

TTelephoneHandsetHandle represents the physical telephone device as opposed to the telephone line. The standard TTelephoneHandsetHandle class has member functions for reporting its hookswitch state and for creating a standard TMicrophone or TSpeaker object representing the handset's audio input or output capability, if any. If such capability is present, the handset will typically operate in two modes, one in which audio is routed directly to and from the physical telephone line, bypassing the computer altogether, and a second in which audio is routed to and from the computer, leaving the physical line out of the loop. Before a handset can be used as a local speaker or microphone, however, it must be explicitly "disconnected" from the physical line. Note that the speaker and microphone created by a TTelephoneHandsetHandle are completely independent of those created by a TTelephoneLineHandle. Playing audio data to the handset's speaker causes sound to come out of the handset. Playing data to the line's speaker causes sound to be sent over the telephone network.

Subclasses may include physical handsets with buttons, displays, or external speakers and microphones with associated gain controls. There is, of course, a "lower" common denominator: a telephone device whose switch status cannot be monitored. In this case, a TTetephoneHandsetHandle object would not be terribly useful. Even when switch status is available, the standard version of TTelephoneHandsetHandle is completely optional for the typical computer-based telephony application.

B. Objects for telephony transactions

In addition to the hardware-type entities associated with use of the telephone network (described above), there are less tangible objects. For example, the connection itself, or, as an end user might think of it, the telephone call could be considered an intangible. A call has an existence and state independent of the telephone line. To make a call a user must first "find" a telephone line. ("Where can I find a telephone? I need to make a call.") To receive a call, a user needs to know which line it is coming in on. ("Mr. Smith, you have a call on line 1.") A line is either on-hook or off-hook, while a call passes through a variety of states during its lifetime. In many cases, a single line can support multiple simultaneous calls where the user has placed one call on hold in order to place a second, outgoing call. Alternatively, a user may add a call to an existing connection to produce a three-way conference call.

In the real world, information about the progress of a call is communicated back to the user via the audio channel: a continuous tone indicates that the call can be dialed, an interrupted tone indicates that the destination line is busy, a "ringing" tone indicates that the destination phone is ringing, etc. In computer-based telephony interfaces these same signals may be obtained, as well as subtler details about a call's state, in the form of alphanumeric status indications.

Telephone Call Handle

As shown in FIG. **9**, between steps **900** and **910**, applications initiate and terminate telephone connections by asking (step **902**) an instance of TTelephoneLineHandle to create a TTelephoneCallHandle (step **904**).

FIG. **10** shows a functional flow for TTelephoneCallHandle. As shown between steps **1000** and **1010**, TTelephoneCallHandle provides access methods for determining the state of the connection (steps **1002** and **1004**) as well as for returning the telephone number of the remote endpoint (step **1006** and **1008**).

Telephone Feature Control

Since TTelephoneLineHandle only permits calls to be placed, answered, and hung up, a family of classes has been provided to facilitate access to the more popular supplementary voice features. In addition to creating calls, a Line Handle can create individual feature control objects for whatever set of advanced features happens to be available. Examples of such features may include the following feature set: Hold, Transfer, Conference, Drop, and Forward. Note that the availability of features depends not only on what the telephony hardware platform is potentially capable of providing, but also on which features the user has actually subscribed for.

Telephone Notification

FIG. **11** is a flow diagram illustrating how an application **1100** sets up to receive certain notifications regarding telephone systems. Clients **1100** wishing to receive any of the notifications discussed below must register interest in them using the standard Notification framework. TInterests **1104** for telephony notifications must be created by calling one of the "create interest" member functions provided by TTelephoneLineHandle and TTelephoneHandsetHandle **1102**.

FIG. **12** illustrates telephony equipment **1200** sending out information via objects which can be used to manage telephony transactions. Information regarding the state of lines, calls, and features is propagated outward from the underlying telephony hardware by means of three telephony-oriented subclasses of TNotification. TTelephoneStatusNotification transports at **1202** an instance of one of three telephony-specific subclasses of TInterest: TTelephoneInterest, TTelephoneCallInterest or TTelephoneFeatureInterest. Status updates themselves are represented by TToken constants. TTelephoneRingNotification propagates information at **1204** about incoming calls. TTelephoneDigitsNotification at **1206** permits clients to receive DTMF digits generated by the remote endpoint.

Phone Numbers

TPhoneNumber, TPhoneEndpoint, and the other classes can be used to represent telephone numbers.

Audio Objects

The TSpeaker and TMicrophone objects returned by TTelephoneLineHandle and TTelephonehandset are subclasses of the speaker and microphone classes. One caveat: data format standards for digital telephony may differ from location to location. Voice data coming in from an ISDN line, for example, will be log-companded rather than linear and digitized at 8 KHz rather than 22 KHz. To address this potential incompatibility, the present invention contemplates format conversion classes. For example, format conversion classes may include: T8KMuLawTo22KConverter, T8KALawTo22KConverter, T22KTo8KMuLawConverter, and T22KTo8KALawConverter.

FIG. **13** provides an example of playing back a sound file using instances of format conversion objects. To play back a sound file originally recorded directly from a North American ISDN line, for example, one would connect the

output port of the file **1300** to the input port of an instance of T8KMuLawTo22KConverter **1302**. One would then connect the output port of the converter to the input port of an instance of TSpeaker **1304** and call the file3 s Play member function. Type negotiation is used by audio ports to select a common data format.

CLASS AND MEMBER FUNCTION DETAILS

This section describes in detail the classes and member functions of the present invention which were introduced above.

As shown in FIG. **14**, to make use of available telephony resources, at **1400** an application instantiates a TTelephoneLineHandle as described above. A telephone connection can then be established by calling the line's CreateAndPlaceCall, represented by **1402**, or CreateAndAnswerCall, represented by **1404**, methods, both of which return a call handle.

To acquire control of an ongoing connection, the line's CreateCallList method would be used, represented by **1406**. Connections initiated in any of these three ways are terminated by calling HangUp on the call handle, represented by **1408**.

TTelephoneLineConfigurationData

This class represents the configuration of a specific telephone line. As shown by FIG. **15**, TTelephoneLineConfigurationData **1500** identifies the directory numbers associated with the line's local endpoint (**1502**) and specifies via type negotiation which telephone features are currently available on the line (**1504**).

TTelephoneLineHandle

TTelephoneLineHandle has been previously discussed with respect to FIGS. **7** and **8**. FIG. **16** shows the functions associated with TTelephoneLineHandle, which is the standard surrogate for a telephone line. It permits selection of a specific line via a telephone line configuration data object (**1600**) and provides member functions for initiating telephone call connections (**1602**), determining the hook status of the line (**1604**), matching telephone feature types via type negotiation (**1606**), and creating audio objects for the line (**1608**). It also generates TInterest subclasses to be used by the notification framework (**1610**) for registering interest in telephony events associated with the line. TTelephoneLineHandle is multi-thread safe, but has not been designed for sharing across team boundaries.

```
class TTelephoneLineConfigurationData:        public MCollectible,
                                              public MTypeNegotiator
//Public constructors
    //Initialize with Collection of Phone Numbers and Sequence of Feature
    //Types, With Phone Numbers Only, or with Single Phone Number
    TTelephoneLineConfigurationData(const Collection& localEndpoint, const
                                              TSequence& featureTypes);
    TTelephoneLineConfigurationData(const TCollection& localEndpoint);
    TTelephoneLineConfigurationData(const TPhoneNumber&);
    //Copy Constructor
    TTelephoneLineConfigurationData(const TTelephoneLineConfigurationData&);
    //Default Constructor for Streaming and Assignment Only
    TTelephoneLineConfigurationData( );
//Destructor
    virtual    ~TTelephoneLineConfigurationData( );
//Assignment Operator TTelephoneLineConfigurationData& operator=(const
    TTelephoneLineConfigurationData&);
//Access Functions
    //Return Endpoint (One or More Phone Numbers) Associated with this Line
            void      GetLocalEndpoint(TPhoneEndpoint&) const;
//MTypeNegotiator Overrides
    //Return List of Active Feature Types
    virtual    void    CreatePreferredTypeList(TSequence&) const;
    //Verify Presence of a Specific Feature Type for Line Represented by
    //this Configuration
    virtual TTypeDescription* ChoosePreferredType(const TSequence&) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneLineConfigurationData);
    virtual    long          Hash( ) const;
    virtual    Boolean       IsEqual(const MCollectible*) const;
    virtual    TStream&      operator<<=(TStream& fromWhere);
    virtual    TStream&      operator>>=(TStream& toWhere) const;
```

```
//Public Constructors
    //Initialize With Configuration Data
    TTelephoneLineHandle(const TTelephoneLineConfigurationData&);
    //Copy Constructor
    TTelephoneLineHandle(const TTelephoneLineHandle&);
    //Default Constructor for Streaming and Assignment Only
    TTelephoneLineHandle( );
//Public Destructor
    virtual  ~TTelephoneLineHandle( );
//Assignment Operator
    virtual TTelephoneLineHandle&       operator=(const TTelephoneLineHandle&);
//Application Interface
    //Return Endpoint Represented by This Line
    virtual  void  GetLocalEndpoint(TCollection&) const;
    //Check Hook Status
    virtual  Boolean  IsOnHook( ) const;
    //Create a Call, Dial Number Specified and Return the Call
    virtual TTelephoneCallHandle* CreateAndPlaceCall(const TPhoneNumber&);
    //Create a Call, Answer an Incoming Call and Return the Call
    virtual TTelephoneCallHandle* CreateAndAnswerCall( );
    //Return Feature Control Object Corresponding to Requested Feature Type
    virtual TTelephoneLineFeatureControl* CreateFeatureControl(const
                                        TTypeDescription& featureType);
    //Return List of TTelephoneCallHandle Objects for
    //All Connections Currently in Progress on the Telephone Line
    virtual void CreateCallList(TCollection&);
    //Send Out Raw DTMF Digits
    virtual  void       SendDigits(const TPhoneCharacters&);
//Audio I/O
    //Create Speaker and Microphone Audio Objects of Appropriate Type
    virtual  TSpeaker*          CreateSpeaker( );
    virtual  TMicrophone*  CreateMicrophone( );
//Notification Framework Support
    //Return a Hook Status Interest for This Line
    virtual  TInterest*         CreateHookInterest( ) const;
    //Return a Ring Notification Interest for This Line
    virtual  TInterest*         CreateRingInterest( ) const;
    //Return a Call Notification Interest for Specified Call Handle
    virtual  TInterest*         CreateCallInterest(const
                                    TTelephoneCallHandle&) const;
    //Return a Feature Notification Interest for Specified Feature Control
    virtual  TInterest*         CreateFeatureInterest(const
                                    TTelephoneLineFeatureControl&) const;
//Low-Level Control Functions
    //Take Line Off-Hook
    virtual  void               TakeOffHook( );\
    //Put Line On-Hook
    virtual  void               PutOnHook( );
        //Dial Fully Specified Phone Number
    virtual  void               DialNumber(const TPhoneNumber&);
    //Send Out Raw DTMF Digits
    virtual  void               SendDigits(const TPhoneCharacters&);
//MTypeNegotiator Overrides
    //Negotiate Availability of Controls for Feature Types
    virtual void CreatePreferredTypeList(TSequence&) const;
    virtual TTypeDescription* ChoosePreferredType(const TSequence&) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneLineHandle);
    virtual     long            Hash ( ) const;
    virtual     Boolean         IsEqual (const MCollectible*) const;
    virtual     TStream&        operator<<= (TStream& fromWhere);
    virtual     TStream&        operator>>= (TStream& toWhere) const;
```

55

TTelephoneCallHandle

This is a lightweight surrogate for a telephone connection. It permits clients to determine the remote number participating in the connection and the current status of the call. TTelephoneCallHandle has been discussed previously with respect to FIGS. **9** and **10**. TTelephoneCallHandle is multi-thread safe, but has not been designed for sharing across team boundaries.

```
class TTelephoneCallHandle : public MCollectible
//Note: This class must be created via a line or line handle
//Destructor
    virtual   ~TTelephoneCallHandle( );
//Client Interface
    //Extract Remote Telephone Number for Connection
    void      GetRemoteNumber(TPhoneNumber& theNum) const;
    //Return Current Call Status
    virtual   TToken    GetStatus( ) const;
    //Hang Up the Call
    virtual   void      HangUp( );
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneCallHandle);
    virtual   long      Hash ( ) const;
    virtual   Boolean   IsEqual (const MCollectible*) const;
//Note: Clients may not assign or stream this class.
```

TTelephoneLineFeatureControl

FIG. 17 shows the creation, at 1700, of TTetephone-LineFeatureControl by TTelephoneLineHandle in response to a client. TTelephoneLineHandles may or may not be capable of supporting arbitrary sets of features above and beyond the ability to make and break telephone connections. For example, a line handle may be able to place a call on hold, transfer a call, conference in a new call, or forward itself to a different line. Clients access features such as these by asking the line handle to create a subclass of TTelephone-LineFeatureControl. Feature control subclasses are specified by means of TTypeDescriptions based on the class names of the desired features. TTelephoneLineFeature-Control is multi-thread safe, but has not been designed for sharing across team boundaries.

```
class TTelephoneLineFeatureControl : public MCollectible
//Static Members for accessing defined Feature Types defined by;
static const void GetHoldFeatureType(TTypeDescription&);
static const void GetDropFeatureType(TTypeDescription&);
static const void GetTransferFeatureType(TTypeDescription&);
static const void GetConferenceFeatureType(TTypeDescription&);
static const void GetForwardFeatureType(TTypeDescription&);
//Note: This class must be created via a line or line handle
//Public Destructor
virtual   ~TTelephoneLineFeatureControl( );
Client Interface
      //Return Feature Type Description
      virtual   void      GetFeatureType(TTypeDescription&) const = 0;
      //Return Current Feature Status
      virtual   TToken    GetStatus( ) const;
//MCollectible Overrides
      MCollectibleDeclarationsMacro(TTelephoneLineFeatureControl);
      virtual   long      Hash ( ) const;
      virtual   Boolean   IsEqual (const MCollectible*) const;
//Note: Clients may not assign or stream this class.
```

Standard TTelephoneLineFeatureControl Subclasses

Five standard feature control subclasses have been provided, corresponding to the "Big Four" supplementary features plus call forwarding.

1) Hold Feature Control:

```
class TTelephoneHoldFeatureControl : public TTelephoneLineFeatureControl
//Public Destructor
virtual       ~TTelephoneHoldFeatureControl( );
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual   void      GetFeatureType(TTypeDescription&) const;
```

-continued

```
//Client Interface
    //Put Currently Active Call On Hold
    virtual   void        PutOnHold( );
    //Recover Specified Call
    virtual   void        Reconnect(const TTelephoneCallHandle&);
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneHoldFeatureControl);
    virtual   long        Hash( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
2) Drop Feature Control:
class TTelephoneDropFeatureControl : public TTelephoneLineFeatureControl
//Public Destructor
virtual       ~TTelephoneDropFeatureControl( );
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual   void        GetFeatureType(TTypeDescription&) const;
//Client Interface
    //Drop Specified Call
    virtual   void        Drop(const TTelephoneCallHandle&);
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneDropFeatureControl);
    virtual   long        Hash( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
3) Transfer Feature Control:
class TTelephoneTransferFeatureControl: public TTelephoneLineFeatureControl
//Public Destructor
    virtual   ~TTelephoneTransferFeatureControl( ),
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual   void        GetFeatureType(TTypeDescription&) const;
//Client Interface
    //Initiate Transfer and Place Call to Specified Number
    virtual TTelephoneCallHandle* CreateCallAndStartTransfer(const
                        TPhoneNumber& newNumber);
    //Finish Ongoing Transfer Operation
    virtual   void        CompleteTransfer( );
    //Abort Ongoing Transfer Operation
    virtual   void        CancelTransfer( );
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneTransferFeatureControl);
    virtual   long        Hash( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
4) Conference Feature Control:
class TTelephoneConferenceFeatureControl: public TTelephoneLineFeatureControl
//Public Destructor
    virtual   ~TTelephoneConferenceFeatureControl( );
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual   void        GetFeatureType(TTypeDescription&) const;
//Client Interface
    //Initiate Conference Operation and Place Call to Specified Number
    virtual TTelephoneCallHandle* CreateCallAndStartAddingToConference(const
                        TPhoneNumber& newNumber);
    //Complete Ongoing Conference Operation
    virtual   void        FinishAddingToConference( );
    //Abort Ongoing Conference Operation
    virtual   void        CancelAddingToConference( );
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneConferenceFeatureControl)
    virtual   long        Hash( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
5) Forward Feature Control:
class TTelephoneForwardFeatureControl : public TTelephoneLineFeatureControl
//Public Destructor
virtual       ~TTelephoneForwardFeatureControl( );
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual   void        GetFeatureType(TTypeDescription&) const;
//Client Interface
    //Forward Line to Specified Number
    virtual   void        Forward(const TPhoneNumber& phoneNumber);
    //Cancel Call Forwarding
    virtual   void        CancelForward( );
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneForwardFeatureControl);
    virtual   long        Hash( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
```

TTelephoneHandsetConfigurationData

This class represents the configuration of a specific telephone handset. It identifies the telephone line or lines to which the handset may be connected. This has been discussed with respect to FIG. **15**.

```
class TTelephoneHandsetConfigurationData : public MCollectible
//Constructors
//Initialize with Configuration(s) of Telephone Lines to Which Handset
//Can be Connected
TTelephoneHandsetConfigurationData(const TCollection&);
TTelephoneHandsetConfigurationData(const TTelephoneLineConfigurationData&);
//Copy Constructor
TTelephoneHandsetConfigurationData(const TTelephoneHandsetConfigurationData&);
//Default Constructor for Streaming and Assignment Only
TTelephoneHandsetConfigurationData( );
//Destructor
virtual        ~TTelephoneHandsetConfigurationData( );
//Assignment Operator
TTelephoneHandsetConfigurationD&          operator=(const
                                          TTelephoneHandsetConfigurationData&);
//Access Function
    //Create Configuration Data Objects for All Telephone Lines to Which
    //the Handset Can Be Connected
    virtual void CreateConnectibleLineConfigurations(TCollection&) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneHandsetConfigurationData);
    virtual   long          Hash ( )const;
    virtual   Boolean             IsEqual(const MCollectible*) const;
    virtual   TStream&    operator<<=(TStream& fromWhere);
    virtual   TStream&    operator>>=(TStream& toWhere) const;
```

TT elephoneHandsetHandle

FIG. **18** provides a functional representation of TTelephoneHandsetHandle, which is the standard surrogate for the physical telephone terminal equipment. It provides a mechanism for monitoring hook status (**1800**) and permits the creation of audio objects (**1802**, **1804**) corresponding to the handset's internal speaker and microphone. It also generates TInterest subclasses to be used by the notification framework (**1806**) for registering interest in telephony events associated with the handset. As a physical device, a handset normally provides manual control for a specific telephone line so that lifting the handset causes some telephone line somewhere to go off hook. Before being used for audio I/O, the handset must be "Disconnected" from its line. It is also possible to connect a handset to one of a set of phone lines. TTelephoneHandsetHandle is multi-thread safe, but has not been designed for sharing across team boundaries.

```
class TTelephoneHandsetHandle : public MCollectible
//Public Constructors
    //Initialize With Configuration Data
    TTelephoneHandsetHandle(const TTelephoneHandsetConfigurationData&);
    //Copy Constructor
    TTelephoneHandsetHandle(const TTelephoneHandsetHandle&);
    //Default Constructor for Streaming and Assignment Only
    TTelephoneHandsetHandle( );
//Public Destructor
virtual        ~TTelephoneHandsetHandle( );
//Assignment Operator
virtual TTelephoneHandsetHandle& operator=(const TTelephoneHandsetHandle&);
//Application Interface
    //Check Hook Status
    virtual   Boolean             IsOnHook( ) const;
    //Check Connection Status
    virtual   Boolean             IsConnected( ) const;
    //Connect to a Specific Phone Line
    //Note: Handset May be Connected to Only One Line at a Time
    virtual void ConnectToTelephoneLine(const TTelephoneLineHandle&);
    //Disconnect from a Specific Phone Line
    virtual void DisconnectFromTelephoneLine( );
    //Create Configuration for Line Currently Connected to This Handset
    virtual TTelephoneLineConfigurationData* CreateLineConfigurationData( )
                                          const;
    //Create Configuration Data Objects for All Lines Available for
    //Connection to this Handset
    virtual void CreateConnectibleLineConfigurations(TCollection&) const;
```

-continued

```
//Audio I/O
    //Create Speaker and Microphone Audio Objects of Appropriate Type
    //Note: These Objects represent the Audio I/O capability of the
    //    Handset itself and not of the line it is connected to
    virtual   TSpeaker*       CreateSpeaker( );
    virtual   TMicrophone*    CreateMicrophone( );
//Notification Framework Support
    //Return a Hook Status Interest for This Handset
    virtual   TInterest*      CreateHookInterest( ) const;
    //Return a Ring Notification Interest for This Handset
    virtual   TInterest*      CreateRingInterest( ) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneHandsetHandle);
    virtual   long            Hash ( ) const;
    virtual   Boolean         IsEqual (const MCollectible*) const;
    virtual   TStream&        operator<<=(TStream& fromWhere);
    virtual   TStream&        operator>>=(TStream& toWhere) const;
```

TInterest Subclasses

FIG. **19** shows TInterest subclasses. The present invention makes use of three specialized TInterest subclasses to communicate status updates: TTelephoneInterest **1900**, TTelephoneCallInterest **1908**, and TTelephoneFeatureInterest **1912**. TTelephoneInterest **1900** permits clients to request notification of certain telephony events affecting a particular telephone line or handset. These events include changes in hook status **1902**, ring indications from incoming calls **1904**, and the arrival of remotely generated digit strings **1906**. TTelephoneCallInterest **1908** permits clients to request notification of changes in the progress of a particular telephone connection **1910**. TTelephoneFeatureInterest **1912** likewise permits clients to request notification of changes in the progress of a particular instance of a telephone feature **1914**. Clients should create instances **1916** of the above classes by calling the appropriate member functions provided by TTelephoneLineHandle and TTelephoneHandsetHandle. None of them is multi-thread safe.

```
• TTelephoneInterest
class TTelephoneInterest : public TInterest
//Public Constructors
TTelephoneInterest(const TTelephoneInterest&);
TTelephoneInterest( );
//Public Destructor
virtual         ~TTelephoneInterest( );
//Assignment Operator
virtual TTelephoneInterest& operator= (const TTelephoneInterest&);
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneInterest);
    virtual   long        Hash ( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
    virtual   TStream&    operator<<=(TStream& fromWhere);
    virtual   TStream&    operator>>=(TStream& toWhere) const;
• TTelephoneCallInterest
class TTelephoneCallInterest : public TInterest
//Public Constructors
TTelephoneCallInterest(const TTelephoneCallInterest&);
TTelephoneCallInterest( );
//Public Destructor
virtual         ~TTelephoneCallInterest( );
//Assignment Operator
virtual TTelephoneCallInterest& operator= (const TTelephoneCallInterest&);
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneCallInterest);
    virtual   long        Hash ( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
    virtual   TStream&    operator<<=(TStream& fromWhere);
    virtual   TStream&    operator>>=(TStream& toWhere) const;
• TTelephoneFeatureInterest
class TTelephoneFeatureInterest : public TInterest
//Public Constructors
TTelephoneFeatureInterest(const TTelephoneFeatureInterest&);
TTelephoneFeatureInterest( );
//Public Destructor
virtual         ~TTelephoneFeatureInterest( );
//Assignment Operator
virtual         TTelephoneFeatureInterest& operator=(const
                                    TTelephoneFeatureInterest&);
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneFeatureInterest);
    virtual   long        Hash ( ) const;
    virtual   Boolean     IsEqual(const MCollectible*) const;
    virtual   TStream&    operator<<=(TStream& fromWhere);
    virtual   TStream&    operator>>=(TStream& toWhere) const;
```

Interest Name Constants

The following public static constants are defined for use with the Notification Framework:

```
• Interest names defined within the scope of TTelephoneInterest
    //Hook Status Update
    static const TToken& kHookStatus;
    //Telephone Ring Interest Name
    static const TToken& kRingNotification;
    //TTelephone Digits Interest Name
    static const TToken& kDigitsReceived;
• Interest names defined within the scope of
TTelephoneCallInterest:
    //Call Status
    static const TToken& kCallStatus;
• Interest names defined within the scope of
TTelephoneFeatureInterest:
    //Telephone Feature Status
    static const TToken& kFeatureStatus;
```

Notification Subclasses

FIG. **20** shows the functional relationships which result from TNotification subclasses. There are three telephony-oriented subclasses of TNotification. TTelephoneStatusNotification **2000** transports a status update **2002** and contains a TTelephoneInterest, a TTelephoneCallInterest, or a TTelephoneFeatureInterest **(2004)** in place of a plain TInterest. Status updates may be assigned to one of four categories, depending upon the progress of the operation with which they are associated: Success, Failure, In Progress, or Unknown. This permits applications to perform a preliminary classification of an incoming notification without explicitly examining its status field. TTelephoneRingNotification **2006** transports the phone number of an incoming call before it has been answered **2008**. Finally, TTelephoneDigitsNotification **2010** transports raw DTMF digits from a remotely connected telephone's keypad **2012**. Clients will typically obtain instances **2014** of these classes via the Notification Framework **2016** rather than by creating instances of their own. These classes are not multi-thread safe.

```
• Status Notification:
class TTelephoneStatusNotification : public TNotification
//Public Constructors
TTelephoneStatusNotification(const TTelephoneStatusNotification&);
TTelephoneStatusNotification(const TInterest&);
//Public Destructor
virtual~TTelephoneStatusNotification( );
//Assignment Operator
virtual    TTelephoneStatusNotification&        operator= (const
           TTelephoneStatusNotification&);
//Client Interface
//Get/Set Status Information
    TToken     GetStatus( ) const;
//Get Notification Classification
    long       GetClassification( ) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneStatusNotification);
    virtual    long          Hash ( ) const;
    virtual    Boolean       IsEqual(const MCollectible*) const;
    virtual    TStream&      operator<<=(TStream& fromWhere);
    virtual    TStream&      operator>>=(TStream& toWhere) const;
• Ring Notification:
class TTelephoneRingNotification : public TNotification
//Public Constructors
TTelephoneRingNotification(const TPhoneNumber&);
TTelephoneRingNotification( );
TTelephoneRingNotification(const TTelephoneRingNotification&);
//Public Destructor
virtual          ~TTelephoneRingNotification( );
//Assignment Operator
virtual          TTelephoneRingNotification&      operator= (const
                                                  TTelephoneRingNotification&);
//Client Interface
    //Extract Calling Party Telephone Number
    void         GetIncomingNumber(TPhoneNumber&) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneRingNotification);
    virtual    long          Hash ( ) const;
    virtual    Boolean       IsEqual(const MCollectible*) const;
    virtual    TStream&      operator<<=(TStream& fromWhere);
    virtual    TStream&      operator>>=(TStream& toWhere) const;
• Incoming Digits Notification:
class TTelephoneDigitsNotification : public TNotification
//Public Constructors
    TTelephoneDigitsNotification(const TTelephoneDigitsNotification&);
    TTelephoneDigitsNotification( );
//Public Destructor
    virtual    ~TTelephoneDigitsNotification( );
//Assignment Operator
virtual          TTelephoneDigitsNotification&    operator= (const
                                                  TTelephoneDigitsNotification&);
//Client Interface
    void         GetDigits(TPhoneCharacters&) const;
//MCollectible Overrides
    MCollectibleDeclarationsMacro(TTelephoneDigitsNotification);
    virtual    long          Hash ( ) const;
    virtual    Boolean       IsEqual(const MCollectible*) const;
    virtual    TStream&      operator<<=(TStream& fromWhere);
    virtual    TStream&      operator>>=(TStream& toWhere) const;
```

Status Constants

The following public static constant TToken references are defined for telephone status updates within the scope of TTelephoneStatusNotification::

Line status constants

kOnHook—Line or handset is on hook.

kOffHook—Line or handset is off hook.

Call status constants

kCallIdle—Call is inactive. The default state.

kIncompleteCallInfo—Call placement is in progress but more information is needed.

kCallReorder—Call placement could not proceed. There was an error in the number.

kCallCannotComplete—Call placement could not be completed.

kReadyToDial—Dial tone is present.

kFastBusy—Dial tone timed out.

kDialingComplete—Dialout completed successfully.

kRingbackDetected—A ringback indication was received. The destination phone is ringing.

kBusyDetected—A busy indication was received. The destination phone number is busy.

kNoAnswer—The destination phone number was not answered.

kCallActive—The call has been successfully connected.

kCallOnHold—The call has been put on hold.

kCallTerminationDetected—The network or the connected party has terminated the call.

Feature control constants

kActivateInProgress—The feature activation request was acknowledged.

kActivateComplete—The feature is now activated.

kCannotActivate—The feature could not be activated.

kDeactivateInProgress—The feature deactivation request was acknowledged.

kDeactivateComplete—The feature is now deactivated.

kCannotDeactivate—The feature could not be deactivated.

kErrorState—An unidentified error occurred.

ETelephones tatusClassification

ETelephoneStatusClassification is an enemy defined within the scope of

TTelephoneStatusNotification: to denote the generic classification of a status notification so that clients can respond appropriately without examining the state value. Defined values are:

kSuccess—Operation Completed Successfully

kFailure—Operation Prevented from Being Completed

kinProgress—Operation Is Proceeding Normally

kUnknown—Progress of Operation is Unknown for this State

Converter Classes

Data format conversion is accomplished via converters such as T8KMuLawTo22KConverter, T8KALawTo22K-Converter, T22KTo8KMuLawConverter, and T22KTo8KALawConverter. These classes inherit from MAudio and add no new interface protocol of their own.

II. Objects for telephony system developers

In addition to the objects discussed above, telephony systems developers will need to have further objects to work with when developing new telephony equipment. For example, third party developers such as AT&T who may wish to add support for proprietary phone equipment and/or to extend the present invention to accommodate advanced telecommunications capabilities. The present invention will make these efforts of third party developers as quick and painless as possible. The telephone architecture should anticipate the need for such extensions while maximizing potential reuse of the existing code associated with standard objects.

Architectural Overview of Objects

Telephony Developers need to concern themselves with three classes which were not described above in the applications section. The abstract Telephone Line class defines a standard interface to telephone network functions. It has a master-surrogate relationship vis-`a-vis TTelephoneLine-Handle and contains a built-in mechanism for communicating with its handles, which may exist in multiple teams. An interface for accessing common supplementary features beyond simple on/off hook and dialing functions is provided by TTelephoneFeature and its subclasses. Developers wishing to provide access to standard functionality on proprietary telephony hardware will need to produce subclasses of the line class and the predefined feature subclasses. Telephone Lines

FIG. **21** represents the abstract base class TTelephoneLine which provides a software model of the physical connection to the telephone network. With each phone line is associated (1) a hook state (on or off) **2100**, (2) a TPhoneEndpoint containing a unique set of Directory Numbers **2102** and (3) a collection of its available telephony features **2104**. In addition, a line manages a list of active handles **2116** to which it directs notification of status updates.

TTelephoneLine contains pure virtual member functions for going offhook, going on-hook, and dialing out digits. Implementations for each of these functions must be provided by the developer. In most cases, such implementation will require communication with an I/O Access Manager and/or a listener task specific to a given hardware/software environment.

If, for example, the physical telephone interface were resident on a nu-bus card running a particular Task A, the role of the device driver might be filled by a TaskACard object instantiated as a member field of the TTelephoneLine subclass. A call to the subclass's OffHook method would cause the line to send a message containing the appropriate command and data to the TaskACard. Responses would be routed back to the line via a listener task containing a TaskAListener. The line subclass would then create the appropriate TNotification subclasses and pass them down to associated TTelephoneLineHandles via the ReceiveUpdate method inherited from TTelephoneLine.

Two additional member functions which may be of interest to a developer are CreateSpeaker, and CreateMicrophone. These two functions have default implementations which simply return a NIL pointer. The request dispatcher, however, is set up to flatten non-NIL pointers for resurrection by the TTetephoneLineHandle on the other side. A preferred embodiment includes audio objects corresponding to the actual telephony hardware and appropriate calls to the sound server to instantiate these objects. Note that in many cases (e.g. a standard external modem connected over the serial port) it will not be possible to pass actual voice data in or out of the host computer and the default implementations will, unfortunately, have to be retained.

It is the responsibility of the TTelephoneLine subclass to ensure that the notification events expected by the line handle and call handle classes are generated in the proper sequence. Telephone Features

In order to provide advanced features beyond those required for setting up and tearing down connections, it is contemplated that the basic TTelephoneLine class support dispatching and processing of requests for advanced telephony features such as "hold", "drop", "transfer", "conference" and "forward", to name five of the more popular ones. The specific set of features available on a given telephone switch depends both upon the inherent capabilities of the switch, and upon the services the switch customer has chosen to subscribe for. Because of this and because of the fact that switch manufacturers come up with newer and fancier features each year, TTelephoneLine operates on a heterogeneous set of features, polymorphically activating and deactivating them.

Continuing with FIG. **21**, the subclasses of TTelephone-LineFeature are shown. TTelephoneLineFeature is the base class from which all features derive and which defines the activation/deactivation protocol. From TTelephoneLineFeature **2104** is derived a set of standard feature subclasses: TTelephoneHoldFeature **2106**, TTelephoneDropFeature **2108**, TTelephoneTransferFeature **2110**, TTelephoneConferenceFeature **2112**, and TTelephoneForwardFeature **2114**. With each of these is associated a type description to be used by any subclass which preserves the feature's control protocol. Standard type descriptions for the five features listed above are provided by TTelephoneLineFeatureControl via static member functions.

A developer who wishes to provide implementations for the standard feature set must create subclasses of TTelephoneHoldFeature **2106**, TTelephoneDropFeature **2108**, TTelephoneTransferFeature **2110**, TTetephoneConferenceFeature **2112** and TTelephoneForwardFeature **2114**, overriding the inherited pure virtual functions Activate and Deactivate so that useful work is performed.

FIG. **22** shows the procedures which are followed if there are advanced features for which no standard TTelephoneLineFeature subclass exists **2202**. At **2204**, if the feature is standard, it is used. If the feature is not standard, it is up to the developer to provide a type description at **2206** by overriding the pure virtual member function GetTypeDescription and to provide implementations of Activate and Deactivate **2208**. The developer must also define a feature control subclass **2210** to present to line handles and must override CreateFeatureControl **2212** so that it returns an instance of the control.

Telephone Configuration

The present invention employs two types of configuration information. Conversion from a displayable to a dialable number, for example, requires that the local area code be known, as well as the prefix for accessing long distance service. Information of this category may vary from location to location, but is common to all telephone lines installed at a given site. The local telephone number, on the other hand, and the features available to the extension associated with that number, are pieces of information which vary from line to line within the same site.

In a corporate setting, site-specific configuration information might be provided by a system administrator and included with the installation package distributed to individual users. Users would still have to specify the line-specific configuration. Non-corporate users would be responsible for providing both sets of information. The actual display and collection of configuration data may require that the developer produce either a special application or a data model capable of collecting input.

Configuration information is represented by TTelephoneLineConfigurationData, a lightweight configuration class which contains all site and line-specific configuration data required by a given TTelephoneLine. A similar relationship obtains between TTelephoneHandsetConfigurationData and TTelephoneHandset. These two classes are described in the preceding section.

Audio Objects

If a developer's TTelephoneLine subclass is capable of capturing and reproducing actual voice signals, it is up to the telephony developer to provide the audio classes required to support this capability via the Sound Server. A developer interested in providing audio I/O from an ISDN card, for example, would write MAudio/TAudioProcessor pairs to represent the input and output capability of that hardware.

Programming Interface for Telephony Developers

This section describes in detail the classes and member functions of the developer's tools introduced in the previous section. Developers who want to implement standard functions and features for a specific telephony environment will subclass TTelephoneLine as well as the standard TTelephoneLineFeature and TTelephoneLineFeatureControl subclasses. Developers who want to add new features and functions will need additional subclasses of TTelephoneLineFeature and TTelephoneLineFeatureControl. Developers who must support specialized terminal equipment will write subclasses of TTelephoneHandset and TTelephoneHandset Handle.

TTelephoneLine

FIG. **23** shows the functional actions of TTelephoneLine **2300**. This is the basic object representing a telephone line. It provides a low-level interface to simple telephone network functionality. Functions include: TakeOffHook **2302**, PutOnHook **2304**, DialNumber **2306**, SendDigits **2308**. The line creates TelephoneCallHandles **2310** to represent connections, obtains TTelephoneFeatureControls **2312** from its set of TTelephoneLineFeatures, and may also be capable of creating microphone and speaker audio objects **2314**. A TTelephoneLine is associated with one or more surrogate objects called TTelephoneLineHandles. Handles relay requests to their master line, which, in turn, distributes telephone hook, ring, call, feature and digit notifications to the line handles. Advanced features such as hold, drop, transfer, conference are supported via subclasses of TTelephoneLineFeature. TTelephoneLine has been designed to be safely shared by handles belonging to different teams.

```
class TTelephoneLine : protected MRemoteDispatcher,
                       public MDelegatingNotification,
                       private MReferenceCounted
//Public Destructor
virtual          ~TTelephoneLine( );
//Public Access Functions
//Specify the Configuration of This Line
//Note: Client May Want to Change Configuration of an Existing Line
virtual void SetConfigurationData(const
                       TTelephoneLineConfigurationData&);
//Create and Return Configuration for This Line
virtual TTelephoneLineConfigurationData* CreateConfigurationData( ) const;
//Protected Constructors
//Specifies Local Configuration Information
TTelephoneLine(const TTelephoneLineConfigurationData&);
//Copy Constructor
TTelephoneLine(const TTelephoneLine&);
//Default Constructor - Use For Streaming Only
TTelephoneLine( );
//Protected Assignment Operator
```

-continued

```
    TTelephoneLine&          operator=(const TTelephoneLine&);
//Subclass Interface
    //Add a Feature Object to the Feature List
    //Line Subclasses Should Create and Adopt Features in Their Constructor
    virtual  void            AdoptFeature(TTelephoneLineFeature*);
    //Specify Feature Type(s) for Which This Line Instance is Configured
    //Must be Called in Constructor After Features have been Adopted
    //Generates Exception of Feature to be configured Does Not Exist
    virtual  void            ConfigureFeatures(const TCollection&);
    //Update Line and Propagate Notification in Response to External Events
    virtual  void            ReceiveUpdate(TNotification&);
    //Setter for Hook Status
    virtual  void            SetHookStatus(const TToken&);
//Process Requests from Handle - These member functions are protected
//Note: Specific implementations for the following 4 pure virtual functions
//    must be provided for actual telephone hardware to be used
    //Take Line Off-Hook
    virtual  void            TakeOffHook( ) =0;
    //Put Line On-Hook
    virtual  void            PutOnHook( ) =0;
    //Dial Fully Specified Phone Number
    virtual  void            DialNumber(const TPhoneNumber&) =0;
    //Send Out Raw DTMF Digits
    virtual  void            SendDigits(const TPhoneCharacters&) =0;
//The Remaining protected request handlers have default implementations:
    //Place Call as Atomic Function - Default Implementation is Provided
    virtual TTelephoneCallHandle*        CreateAndPlaceCall(const TPhoneNumber&);
    //Answer Call as Atomic Function - Default Implementation is Provided
    virtual TTelephoneCallHandle*        CreateAndAnswerCall( );
    //Return Handles for All Calls In Progress on This Line
    virtual  void            CreateCallList(TCollection&) const;
    //Return Current Hook Status
    virtual  TToken          UpdateHookstatus( ) const;
    //Return Current Status of Specified Call
    virtual  TToken          UpdareCallStatus(const TTelephoneCallHandle&);
    //Return Feature Control Object Corresponding to Requested Feature Type
    virtual TTelephoneLineFeatureControl* CreateFeatureControl(
const TTypeDescription&) const;
//Protected Audio I/O Functions
    //Create Speaker and Microphone Audio Objects of appropriate type
    virtual  TSpeaker*       CreateSpeaker( ) = 0;
    virtual  TMicrophone*    CreateMicrophone( ) = 0;
//Protected Call Management Functions
    //Create a New Connection
    virtual  TTelephoneCallHandle*    AddCallAndCreateHandle(
                                        const TPhoneNumber&);
    //Locate Handle for Call Which Matches Specified State
    virtual  TTelephoneCallHandle*    FindCallAndCreateHandle(
                                        const TToken& callState) const;
    //Eliminate an Existing Connection
    virtual  void          DeleteCall(const TTelephoneCallHandle&);
//Protected Feature Management Functions
    //Locate Feature of Specified Type
    virtual TTelephoneLineFeature* GetFeature(
                        const TTypeDescription& featureType) const;
//Protected Functions for Generating Notifications
    virtual TNotification* CreateHookNotification(const TToken& hookState,
TTelephoneStatusNotification::ETelephoneStatusClassification
        classification =TTelephoneStatusNotification::kSuccess) const;
    virtual TNotification* CreateRingNotification(const TPhoneNumber&)
                                        const;
    virtual TNotification* CreateCallNotification(const TToken& callState,
        const TTelephoneCallHandle&,
        TTelephoneStatusNotification::ETelephoneStatusClassification
        classification =TTelephoneStatusNotification::kSuccess) const;
    virtual TNotification* CreateFeatureNotification(const TToken&
        featureState, const TTelephoneLineFeature&,
        TTelephoneStatusNotification::ETelephoneStatusClassification
        classification =TTelephoneStatusNotification::kSuccess) const;
    virtual TNotification* CreateDigitsNotification(const TPhoneCharacters&)
                                        const;
//Public MCollectible Overrides
    VersionDeclarationsMacro(TTelephoneLine);
    virtual  long            Hash( ) const;
    virtual  Boolean         IsEqual(const MCollectible*) const;
//Protected Streaming Operators Should Only Be Called By Subclasses
    virtual  TStream&        operator<<=(TStream& fromWhere),
    virtual  TStream&        operator>>=(TStream& toWhere) const;
```

TTelephoneHandset

FIG. 24 shows the functional flows performed by TTelephoneHandset **2400**. This is the basic object representing a physical telephone handset. It may also be capable of creating microphone and speaker audio objects **2402**. A TTelephoneHandset **2400** is associated with one or more surrogate objects called TTelephoneHandsetHandles **2404**. Handles relay requests **2406** to their master handset, which, in turn, distributes status updates **2408** to handles. TTelephoneHandset **2400** has been designed to be safely shared by handles belonging to different teams.

```
class TTelephoneHandset : protected MRemoteDispatcher,
                          public MDelegatingNotifier,
                          public MReferenceCounted
//Public Destructor
virtual     ~TTelephoneHandset( );
//Public Access Functions
//Specify the Configuration of This Handset
//Note: Client May Want to Change Configuration of an Existing Line
virtual void SetConfigurationData(const TTelephoneHandsetConfigurationData&);
//Create and Return Configuration for This Handset
virtual TTelephoneHandsetConfigurationD* CreateConfigurationData( ) const;
//Protected Constructors
//Specifies Local Configuration Information
TTelephoneHandset(const TTelephoneHandsetConfigurationData&);
//Copy Constructor
TTelephoneHandset(const TTelephoneHandset&);
//Default Constructor - Use For Streaming Only
TTelephoneHandset( );
//Assignment Operator
TTelephoneHandset& operator=(const TTelephoneHandset&);
//Sub-Class Interface
    //Update Line and Propagate Notification in Response to External Events
    virtual  void          ReceiveUpdate(TNotification&);
    //Protected Setter for Hook Status
    virtual  void          SetHookStatus(const TToken& theStatus);
    //Locate Connectable Telephone Line Using Specified Configuration
    //This function has been provided to facilitate implementation of the
    //"ConnectToTelephoneLine" pure virtual function
    virtual  TTelephoneLine* FindLine(
                           const TTelephoneLineConfigurationData&);
//Process Requests from Handle - These Member Functions are Protected
//Note: Specific implementations for the following 2 pure virtual functions
//    must be provided for actual telephone hardware to be used
    //Connect to Line Specified by Configuration Data
    //Connectability Verification, Connection State Maintenance Done
    //Automatically
    virtual void ConnectToTelephoneLine(const TTelephoneLineHandle&) = 0;
    //Disconnect from Currently Associated Line
    //Connection State Maintained Automatically
    virtual void DisconnectFromTelephoneLine( ) =0;
//The Remaining protected request handlers have default implementations:
    //Getter for Hook Status
    virtual  TToken        UpdateHookStatus( ) const;
    //Getter for Connection Status
    virtual  Boolean       UpdateConnectionStatus( ) const;
    //Get Configuration Data for Currently Connected Telephone Line
    //At Any Time, Handset may be Connected to Zero or One of These Lines
    virtual TTelephoneLineConfigurationData* CreateLineConfigurationData( )
                                                        const;
    //Create Configuration Data Objects for All Telephone Lines to Which
    //Handset Can Be Connected
    virtual void CreateConnectibleLineConfigurations(TCollection&) const;
//Protected Audio I/O Functions
    //Create Speaker and Microphone Audio Objects of appropriate type
    //Note: These Objects Represent the Local Audio I/O Capability of the
    //Handset and Not of the Telephone Line to Which it is Connected
    virtual  TSpeaker*     CreateSpeaker( ) =0;
    virtual  TMicrophone*  CreateMicrophone( ) =0;
//Protected Functions for Generating Notifications
    virtual TNotification* CreateHookNotification(const TToken& hookState,
    TTelephoneStatusNotification::ETelephoneStatusClassification
        classification =TTelephoneStatusNotification::kSuccess) const;
    virtual TNotification* CreateRingNotification(const TPhoneNumber&)
                                                        const;
//Public MCollectible Overrides
    VersionDeclarationsMacro(TTelephoneHandset);
    virtual  long          Hash ( ) const;
    virtual  Boolean       IsEqual (const MCollectible*) const;
//Protected Streaming Operators Should Only Be Called By Subclasses
    virtual  TStream&      operator<<=(TStream& fromWhere);
    virtual  TStream&      operator>>=(TStream& toWhere) const;
```

FIG. 25 shows the functions of TTelephoneLineFeature **2500**. TTelephoneLineFeature is a base class which defines a standard set of advanced Telephone Line features. It defines a protocol for polymorphically Activating and Deactivating **2504** advanced telephony features **2510** and provides methods for determining their type **2506** and querying their status fields **2508**. TTelephoneLineFeature is not multi-thread safe, but is accessed in a safe manner by the Line.

```
class TTelephoneLineFeature : protected MRemoteDispatcher
//Public Destructor
    virtual   ~TTelephoneLineFeature( );
//Public Access Functions
    //Return Feature Type Description - Pure Virtual Function
    virtual   void              GetFeatureType(TTypeDescription&) const =0;
    //Return Current Feature Status
    virtual   TToken            UpdateStatus( ) const;
    //Create Matching Control Object
    //Pure Virtual Member Function - Must Be Overridden
    virtual TTelephoneLineFeatureControl* CreateFeatureControl( ) const =0;
//Protected Constructors
    //For Use By Subclasses - Defaults to Deactivated State
    TTelephoneLineFeature( );
    //Copy Constructor
    TTelephoneLineFeature(const TTelephoneLineFeature&);
//Protected Assignment Operator
    TTelephoneLineFeature&      operator=(const TTelephoneLineFeature&);
//Subclass Interface
    //Activate/Deactivate the Feature Polymorphically
    //Pure Virtual Functions - Must Be Overridden
    virtual   void              Activate( ) =0;
    virtual   void              Deactivate( ) =0;
//Public MCollectible Overrides
    VersionDeclarationsMacro(TTelephoneLineFeature);
    virtual   long              Hash( ) const;
    virtual   Boolean                 IsEqual(const MCollectible*) const;
//Protected Streaming Operators Should Only Be Called By Subclasses
    virtual   TStream&          operator<<=(TStream& fromWhere);
    virtual   TStream&          operator>>=(TStream& toWhere) const;
```

• Hold Feature:

```
class TTelephoneHoldFeature : public TTelephoneLineFeature
//Destructor
    virtual   ~TTelephoneHoldFeature( );
//TTelephoneLineFeature Overrides
    //Return Standard Hold Type Description
    virtual   void          GetFeatureType(TTypeDescription&) const;
    //Return Standard Hold Control Object
    virtual   TTelephoneFeatureControl* CreateFeatureControl( ) const;
//Protected Constructors
    TTelephoneHoldFeature( );
    TTelephoneHoldFeature(const TTelephoneHoldFeature&);
```

• Drop Feature:

```
class TTelephoneDropFeature : public TTelephoneLineFeature
//Public Destructor
    virtual   ~TTelephoneDropFeature( );
//TTelephoneLineFeature Overrides
    //Return Standard Drop Type Description
    virtual   void          GetFeatureType(TTypeDescription&) const;
    //Return Standard Drop Control Object
    virtual   TTelephoneFeatureControl* CreateFeatureControl( ) const;
//Protected Constructors
    TTelephoneDropFeature( )
    TTelephoneDropFeature(const TTelephoneDropFeature&);
```

• Transfer Feature:

```
class TTelephoneTransferFeature : public TTelephoneLineFeature
//Destructor
    virtual   ~TTelephoneTransferFeature( );
//TTelephoneLineFeature Overrides
```

-continued

```
    //Return Standard Transfer Type Description
    virtual    void          GetFeatureType(TTypeDescription&) const;
    //Return Standard Transfer Control Object
    virtual    TTelephoneFeatureControl* CreateFeatureControl( ) const;
//Protected Constructors
    TTelephoneTransferFeature( );
    TTelephoneTransferFeature(const TTelephoneTransferFeature&);
• Conference Feature:
class TTelephoneConferenceFeature : public TTelephoneLineFeature
Destructor
    virtual    ~TTelephoneConferenceFeature( );
//TTelephoneLineFeature Overrides
    //Return Standard Conference Type Description
    virtual    void          GetFeatureType(TTypeDescription&) const;
    //Return Standard Conference Control Object
    virtual    TTelephoneFeatureControl* CreateFeatureControl( ) const;
//Protected Constructors
    TTelephoneConferenceFeature( );
    TTelephoneConferenceFeature(const TTelephoneConferenceFeature&);
• Forward Feature:
class TTelephoneForwardFeature : public TTelephoneLineFeature
//Public Destructor
    virtual    ~TTelephoneForwardFeature( );
//TTelephoneLineFeature Overrides
    //Return Standard Forward Type Description
    virtual    void          GetFeatureType(TTypeDescription&) const;
    //Return Standard Forward Control Object
    virtual TTelephoneFeatureControl* CreateFeatureControl( ) const;
//Protected Constructors
    TTelephoneForwardFeature( );
    TTelephoneForwardFeature(const TTelephoneForwardFeature&);
```

TTelephoneLineHandle

The constructor, destructor, and some other member functions of this class have already been described in the section on the interface for application writers. The following low-level control functions provided by TTelephoneLineHandle may be useful when authoring a TTelephoneLineFeature-Control subclass.

```
    //Take Line Off-Hook
    virtual    void          TakeOffHook( );
    //Put Line On-Hook
    virtual    void          PutOnHook( );
    //Dial Fully Specified Phone Number
    virtual    void          DialNumber(const TPhoneNumber&);
    //Send Out Raw DTMF Digits
    virtual    void          SendDigits(const TPhoneCharacters&);
```

TTelephoneFeatureControl

The following is a similar list of functions for TTelephoneFeatureControl. These are used internally to implement the feature-specific protocol of the control. The pointer returned by GetLineHandle gives the control access to the public member functions of its associated line handle.

```
    //Send Activation/Deactivation Request to Corresponding Feature Object
    virtual        void          Activate( );
    virtual        void          Deactivate( );
    //Return Reference to Call's Line Handle
    TTelephoneLineHandle*    GetLineHandle( ) const;
```

Audio Considerations

Interaction with the public telephone network itself is typically intermittent and relatively slow, thus imposing few, if any, strict performance constraints. Presence of audio data, on the other hand, brings in the standard set of real-time requirements that go along with analog-to-digital and digital-to-analog conversion hardware. Failure to keep up with the demands of such hardware introduces undesirable data discontinuities which not only degrade the intelligibility of speech data but can be extremely unpleasant (if not painful) to listen to. These real-time requirements, however, should not compromise the performance of non-real-time processes such as user interface widgets.

Specifically, recording or playing back one channel of audio does not provide any noticeable adverse effects upon such standard user operations as (a) selecting a menu item, (b) typing a character, (c) selecting a graphic object or (d) a range of text. This should be possible under each of the following scenarios:

1)8 KHz log-companded samples captured from telephone handset and recorded to disk

2)8 KHz log-companded samples captured from telephone network and recorded to disk

3)8 KHz log-companded samples retrieved from disk and played back to handset

4)8 KHz log-companded samples retrieved from disk and played back to network

5)8 KHz log-companded samples retrieved from disk and played back to system speaker

6)22 KHz linear samples retrieved from disk and played back to handset

7)22 KHz linear samples retrieved from disk and played back to network

Scenarios 5 through 7 require sample rate conversion with simultaneous companded-to-linear or linear-to-companded conversion.

In order for the present invention to operate with audio objects, the telephony objects support a variety of standards. For example, it is contemplated that both the μ-Law and the A-Law formats as well as providing for conversion to and from the standard audio format(s) is supported by target machines. By providing the capability of interacting with audio objects, the boundary between the telephony world and the generic sound world appears smooth and "seamless". It would be possible to instantiate via the Sound Server a standard microphone or speaker object which encapsulates the audio input or output capabilities of telephone lines and telephone handsets.

EXAMPLES in Accordance With A Preferred Embodiment

The above discussion provides a description of some exemplary objects which may be useful for interfacing with telephony equipment. Below provides examples using some of the objects discussed above.

```
Placing a Call:
. . .
//Create a New TTelephoneHandle.
//Line Configuration is a TTelephoneHandsetConfigurationData object
TTelephoneLineHandle myLineHandle(lineConfiguration);
//Create a New Call Object
TPhoneNumber myNumber =TPhoneNumber("974-0001")
TTelephoneCallHandle* myPhoneCall =myLineHandle.CreateAndPlaceCall(myNumber);
. . .
//OK, time to get off the phone
myPhoneCall.HangUp( );
delete myPhoneCall;
Answering a Call:
. . .
//For the Sake of the Example We'll Declare A Special Class
//to Receive the Ring Notification
class TRingHandler
{
public:
    TRingHandler (const TTelephoneLineHandle*);
virtual    ~TRingHandler( );
virtual    void        HandleRing(const TTelephoneRingNotification&);
private:
TTelephoneLineHandle*              fLineHandle;
TTelephoneCallHandle*              fCallHandle;
TMemberFunctionConnection          fConnection,
}
//The Definition of TRingHandler Would Then Be As Follows
TRingHandler::TRingHandler(const TTelephoneLineHandle* lineHandle)
                    : fLineHandle(lineHandle)
{
    fConnection.SetReceiver(this, (NotificationMemberFunction)
            &TRingHandler::ReceiveEvent);
        fConnection.AdoptInterest(fLineHandle->CreateRingInterest( ));
        fConnection.Connect( );
}
TRingHandler::~TRingHandler( )
{ }
TRingHandler::HandleRing(const TNotification& ringNotification)
{
        //Before the Call is Connected, We Can Ask Who Is Calling
        TPhoneNumber callingParty;
        ((const TTelephoneRingNotification&)
                theNotification).GetIncomingNumber(callingParty);
        //Go Ahead and Answer It
        fCallHandle =fLineHandle->CreateAndAnswerCall( )
}
Putting a Call On Hold:
. . .
//Create a New TTelephoneHandle as Above
TTelephoneLineHandle myLineHandle(lineConfiguration);
//Negotiate for Hold Feature
//In Real Life, This Would Be Done with Surrogate Before Instantiating Line
TDeque featureList;
TTypeDescription holdFeatureType;
TTelephoneLineFeatureControl::GetHoldFeatureType(holdFeatureType)
featureList.AddLast(&holdFeatureType);
TTypeDescription* featureType =myLineHandle.ChoosePreferredType(featureList);
Boolean canHold =(featureType !=NIL);
//Create a New Call Object
```

-continued

```
TPhoneNumber myNumber =TPhoneNumber("974-0001")
TTelephoneCallHandle* myPhoneCall =myLineHandle.CreateAndPlaceCall(myNumber);
. . .
//OK, the Call's Now Active So Put It On Hold
if ((myPhoneCall->GetStatus( ) ==TTelephoneStatusNotification::kCallActive)
        && (canHold)) {
    TTelephoneFeatureControl* holdControl;
    holdControl =myLineHandle.CreateFeatureControl(holdFeatureType);
    holdControl->PutOnHold( );
. . .
    //He's Suffered Long Enough. Let's Reconnect Him.
    holdControl->Reconnect(myPhoneCall);
    }
. . .
Adding a New Telephone Feature:
// ---------------------------------------------------------------
//    TFancyFeature - Declaration
// ---------------------------------------------------------------
class TFancyFeature : public TTelephoneLineFeature {
public:
//Constructors
    TFancyFeature( );
    TFancyFeature(const TFancyFeature&);
//Destructor
    virtual    ~TFancyFeature( );
//Assignment Operator
    TFancyFeature&      operator=(const TFancyFeature&);
//TTelephoneLineFeature Overrides
    //Return Fancy Feature Type Description
    virtual    void             GetFeatureType(TTypeDescription&) const;
    //Return Fancy Feature Control Object
    virtual    TTelephoneFeatureControl* CreateFeatureControl( ) const;
    //Activate Feature
    virtual    void             Activate( );
    //Deactivate Feature
    virtual    void             Deactivate( );
//MCollectible Overrides
MCollectibleDeclarationsMacro(TFancyFeature);
    virtual    long             Hash( ) const;
    virtual    Boolean          IsEqual(const MCollectible*) const;
    virtual    TStream&         operator<<=(TStream& fromWhere);
    virtual    TStream&         operator>>=(TStream& toWhere) const;
}
// ---------------------------------------------------------------
//    MFancyFeature - Definition
// ---------------------------------------------------------------
//Constructor implementation
TFancyFeature::TFancyFeature( )
{ }
//Destructor Implementation
TFancyFeature::~TFancyFeature( )
{ }
//Return Fancy Feature Type Description
void          GetFeatureTypc(TTypeDescription& featureType) const
{
    TFancyFeatureControl::GetFancyFeatureType(featureType);
}
//Create Fancy Feature Control
TTelephoneFeatureControl* TFancyFeature::CreateFeatureControl( ) const
{
    TTelephoneFeatureControl* fancyControl =new TFancyFeatureControl( );
    return fancyControl;
}
//Activate Feature
void             TFancyFeature::Activate( )
{
    //Do Actual Work to Activate Feature
}
//Deactivate Feature
void             TFancyFeature::Deactivate( )
{
    //Do Actual Work to Deactivate Feature
}
//MCollectible Overrides
MCollectibleDefinitionsMacro(TFancyFeature,0);
. . .
// ---------------------------------------------------------------
//    TFancyFeatureControl - Declaration
// ---------------------------------------------------------------
```

-continued

```
class TFancyFeatureControl: public TTelephoneFeatureControl {
public:
static const void GetFancyFeatureType(TTypeDescription&);
//Destructor
    virtual         ~TFancyFeatureControl( );
//TTelephoneLineFeatureControl Override
    //Return Feature Type Description
    virtual  void        GetFeatureType(TTypeDescription&) const;
//Client Interface
    //Access the Feature - May Need to Pass Line or Call Handle Parameter
    virtual  void        TuurnOnFancyFeature( );
    virtual  void        TurnOffFancyfeature( );
    //MCollectible Overrides
    MCollectibleDeclarationsMacro(TFancyFeatureControl);
    virtual  long        Hash( ) const;
    virtual  Boolean     IsEqual(const MCollectible) const;
protected:
    virtual  TStream&    operator<<=(TStream& fromWhere);
    virtual  TStream&    operator>>=(TStream& toWhere) const;
//Constructors
    //Initialize with Values From Specified Feature
    TFancyFeatureControl(const TFancyFeatureControl&);
    //Copy Constructor
    TFancyFeatureControl(const TFancyFeatureControl&);
    //Default Constructor Called by TFancyFeature
    TFancyFeatureControl( );
    //Assignment Operator
    TFancyFeatureControl& operator=(const TFancyFeatureControl&);
//Only a TFancyFeature Can Construct This Class
    friend class TFancyFeature;
};
// //-----------------------------------------------------------
//    TFancyFeatureControl - Definition
// //-----------------------------------------------------------
const void TTelephoneLineFeatureControl::GetFancyFeatureType
                        (TTypeDescription& featureType)
{
    static const TTypeDescription fancyType("BrandXFancyFeature");
    featureType =fancyType;
}
//Destructor
TFancyFeatureControl::~TFancyFeatureControl( )
//Return Feature Type Description
void        TFancyFeatureControl::GetFeatureType(TTypeDescription& featureType) const
{
    GetFancyFeatureType(featureType);
}
//Client Interface
void        TFancyFeatureControl::TurnOnFancyFeature( )
{
    Activate( );
}
void        TFancyFeatureControl::TurnOffFancyFeature( )
{
    Deactivate( );
}
//MCollectible Overrides
MCollectibleDefinitionsMacro(TFancyFeatureControl,0);
//Constructors
//Initialize with Values From Specified Feature
TFancyFeatureControl::TFancyFeatureControl(const TFancyFeature& theFeature)
        :TTelephoneLineFeatureControl(theFeature)
{
}
//Copy Constructor
TFancyFeatureControl::TFancyFeatureControl(const TFancyFeatureControl& source)
        :TTelephoneLineFeatureControl(source)
{
}
//Default Constructor Called for Streaming and Assignment Only
TFancyFeatureControl::TFancyFeatureControl( ) : TTelephoneLineFeatureControl( )
{
}
//Assignment Operator
TFancyFeatureControl&    TFancyFeatureControl::operator= (
                            const TFancyFeatureControl& source)
{
    //No Self-Assignment
    if (&source !=this) {
```

-continued

```
    TTelephoneFeatureControl:operator=(source);
    }
  return *this; }
```

Predefined Telephone Constants

Name	Meaning
Telephone Notification Interest Names	
TTelephoneInterest::kHookStatus	Hook Status Update
TTelephoneCallInterest::kCallStatus	Call Status Update
TTelephoneFeatureInterest::kFeatnreStatus	Feature Status Update
TTelephoneInterest::kRingNotification	Notification of Incoming Call
TTelephoneInterest::kDigitsReceived	Notification of Incoming DTMF
Telephone Line States	
TTelephoneStatusNotification::kOnHook	Line is On Hook
TTelephoneStatusNotification::kOffHook	Line is off Hook
Telephone Call States	
TTelephoneStatusNotification::kCallIdle	Default State
TTelephoneStatusNotification::kIncompleteCallInfo	More Information Needed to Place Call
TTelephoneStatusNotification::kCallReorder	Call Placement Could Not Proceed
TTelephoneStatusNotification::kCallCannotComplete	Special Service Information Tone Detected
TTelephoneStatusNotification::kReadyToDial	Dialtone is Present
TTelephoneStatusNotification::kFastBusy	Dialtone Timeout
TTelephoneStatusNotification::kDialingComplete	Dialout Completed
TTelephoneStatusNotification::kRingbackDetected	Remote Connection is "ringing"
TTelephoneStatusNotification::kBusyDetected	Busy Received - Remote Phone is Busy
TTelephoneStatusNotification::kNoAnswer	Remote Phone has not been Answered
TTelephoneStatusNotification::kCallActive	Call has been Successfully Connected
TTelephoneStatusNotification::kCallOnHold	Call has been Put on Hold
TTelephoneStatusNotification::kCallTerminationDetected	Other Party has Terminated
Telephone Feature States	
TTelephoneStatusNotification::kActivateInProgress	Feature Activation Request Acknowledged
TTelephoneStatusNotification::kActivateComplete	Feature is Now Activated
TTelephoneStatusNotification::kCannotActivate	Feature Could Not Be Activated
TTelephoneStatusNotification::kDeactivate	Feature Deactivation Request Ack InProgress
TTelephoneStatusNotification::kDeactivateComplete	Feature is Now Deactivated
TTelephoneStatusNotification::kCannotDeactivate	Feature Could Not Be Deactivated
Generic Error State	
TTelephoneStatusNotification::kErrorState	Bad or Undefined State

Basic Telephone Call State Machine

The following provides an implementation of a call state machine utilizing some of the objects discussed above.

```
Start State: kCallIdle [Outbound Call]
        Action: TTelephoneLineHandle::TakeOffHook        EndState: kReadyToDial
Start State: kCallIdle [Incoming Call]
        Action: TTelephoneLineHandle::TakeOffHook        EndState: kCallActive
Start State: kReadyToDial
        Event: kFastBusyDetected                         EndState: kFastBusyDetected
        Action: TTelephoneLineHandle::SendDigits         EndState: kDialingComplete
        Action: TTelephoneLineHandle::SendDigits         EndState: kIncompleteCallInfo
        Action: TTelephoneLineHandle::SendDigits         EndState: kCallReorder
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kFastBusyDetected
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kDialingComplete
        Event: kRingbackDetected                         EndState: kRingbackDetected
        Event: kBusyDetected                             EndState: kBusyDetected
        Event: kCallCannotComplete                       EndState: kCallCannotComplete
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kIncompleteCallInfo
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kCallReorder
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kRingbackDetected
        Event: kCallActive                               EndState: kCallActive
        Event: kNoAnswer                                 EndState: kNoAnswer
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kBusyDetected
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: kCallCannotComplete
        Action: TTelephoneCallHandle::HangUp             EndState: kCallIdle
Start State: TTelephoneLine::kCallActive
        Event: kCallIdle                                 EndState: kCallIdle
```

-continued

Action: TTelephoneCallHandle::HangUp	EndState: kCallIdle
Start State: TTelephoneLine::kNoAnswer	
Action: TTelephoneCallHandle::HangUp	EndState: kCallIdle

While the invention has been described in terms of a preferred embodiment in a specific system environment, those skilled in the art recognize that the invention can be practiced, with modification, in other and different hardware and software environments within the spirit and scope of the appended claims.

Having thus described our invention, what we claim as new, and desire to secure by Letters Patent is:

1. A telephony apparatus, comprising:

(a) a processor;

(b) a storage attached to and controlled by the processor;

(c) an object oriented operating system, supporting encapsulation, polymorphism and inheritance, including objects, each of the objects containing logic and data resident in the storage and controlling operations of the processor;

(d) a display attached to the processor under the control of the object oriented operating system;

(e) a telephony element attached to the processor;

(f) a telephony object, including logic for interfacing the telephony element to the processor and data for storing status information associated with the telephony element in the telephony object, and representative of the telephony element under the control of the object-oriented operating system, stored in the storage and displayed on the display; and

(g) means for controlling the telephony element by the object oriented operating system utilizing the logic in the telephony object to interface the telephony element to the processor by initiating a call connection, monitoring call progress, activating call featurs and storing status information in the data of the telephony object.

2. The apparatus as recited in claim 1, including means for translating information received from the telephony element into information the object oriented operating system can utilize.

3. The apparatus as recited in claim 1, including means for translating information received from the telephony object into information the telephony element can utilize.

4. The apparatus as recited in claim 1, including means for attaching the telephony element to the processor.

5. The apparatus as recited in claim 4, including means for connecting a telephone line to the processor.

6. The apparatus as recited in claim 4, including means for connecting a handset to the processor.

7. The apparatus as recited in claim 4, including means for setting up a call to the processor.

8. The apparatus as recited in claim 4, including means for passing information between the telephony element and the processor.

9. The apparatus as recited in claim 8, including means for exchanging DTMF tones between the telephony element and the processor.

10. The apparatus as recited in claim 1, including means for enabling features of the telephony element via the telephony object.

11. The apparatus as recited in claim 1, including means for servicing queries between a telephony element and the object-oriented operating system.

12. The apparatus as recited in claim 1, including means for exchanging notification information between a telephony element and the object-oriented operating system.

13. A method for enabling telephony elements on a computer system, including a processor with an attached storage, display and telephony element, comprising the steps of:

(a) controlling operations of the processor with an object oriented operating system, supporting encapsulation, polymorphism and inheritance, including objects, each of the objects containing logic and data resident in the storage;

(b) creating a telephony object, including logic for interfacing the telephony element to the processor and data for storing status information associated with the telephony element in the telephony object, and representative of the telephony element under the control of the object-oriented operating system, stored in the storage and displayed on the display; and

(c) controlling the telephony element by the object-oriented operating system utilizing logic in the telephony object to interface the telephony element to the processor by initiating a call connection, monitoring call progress, activating call features and storing status information in the data of the telephony object.

14. The method as recited in claim 13, including the step of translating information received from the telephony element into information the object oriented operating system can utilize.

15. The method as recited in claim 13, including the step of translating information received from the telephony object into information the telephony element can utilize.

16. The method as recited in claim 13, including the step of attaching the telephony element to the processor.

17. The method as recited in claim 16, including the step of connecting a telephone line to the processor.

18. The method as recited in claim 16, including the step of connecting a handset to the processor.

19. The method as recited in claim 16, including the step of setting up a call to the processor.

20. The method as recited in claim 16, including the step of passing information between the telephony element and the processor.

21. The method as recited in claim 20, including the step of exchanging DTMF tones between the telephony element and the processor.

22. The method as recited in claim 13, including the step of enabling features of the telephony element via the telephony object.

23. The method as recited in claim 13, including the step of exchanging status information between a telephony element and the object-oriented operating system.

24. The method as recited in claim 13, including the step of exchanging notification information between a telephony element and the object-oriented operating system.

* * * * *